戦略シミュレーションゲームの作り方
How to programming a tactical simulation game
GAMKIN株式会社
ロバート・ジェイ・ゴールド 著
SE
SHOEISHA
Cover Illustration: たえ☆ぽん

戦略
シミュレーションゲームの作り方
How to programming a tactical simulation game
GAMKIN株式会社
ロバート・ジェイ・ゴールド 著
SE
SHOEISHA

本書内容に関するお問い合わせについて

このたびは翔泳社の書籍をお買い上げいただき、誠にありがとうございます。弊社では、読者の皆様からのお問い合わせに適切に対応させていただくため、以下のガイドラインへのご協力をお願い致しております。下記項目をお読みいただき、手順に従ってお問い合わせください。

●ご質問される前に

弊社Webサイトの「正誤表」をご参照ください。これまでに判明した正誤や追加情報を掲載しています。

正誤表　http://www.shoeisha.co.jp/book/errata/

●ご質問方法

弊社Webサイトの「刊行物Q&A」をご利用ください。

刊行物Q&A　http://www.shoeisha.co.jp/book/qa/

インターネットをご利用でない場合は、FAXまたは郵便にて、下記"翔泳社 愛読者サービスセンター"までお問い合わせください。電話でのご質問は、お受けしておりません。

●回答について

回答は、ご質問いただいた手段によってご返事申し上げます。ご質問の内容によっては、回答に数日ないしはそれ以上の期間を要する場合があります。

●ご質問に際してのご注意

本書の対象を越えるもの、記述個所を特定されないもの、また読者固有の環境に起因するご質問等にはお答えできませんので、あらかじめご了承ください。

●郵便物送付先およびFAX番号

送付先住所　〒160-0006　東京都新宿区舟町５

FAX番号　03-5362-3818

宛先　（株）翔泳社 愛読者サービスセンター

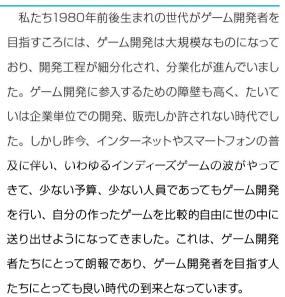
※本書に記載されたURL等は予告なく変更される場合があります。

※本書の出版にあたっては正確な記述につとめましたが、著者や出版社などのいずれも、本書の内容に対してなんらかの保証をするものではなく、内容やサンプルに基づくいかなる運用結果に関してもいっさいの責任を負いません。

※本書に掲載されているサンプルプログラムやスクリプト、および実行結果を記した画面イメージなどは、特定の設定に基づいた環境にて再現される一例です。

刊行にあたって

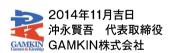
GAMKIN株式会社
代表取締役　沖永賢吾

　私たち1980年前後生まれの世代がゲーム開発者を目指すころには、ゲーム開発は大規模なものになっており、開発工程が細分化され、分業化が進んでいました。ゲーム開発に参入するための障壁も高く、たいていは企業単位での開発、販売しか許されない時代でした。しかし昨今、インターネットやスマートフォンの普及に伴い、いわゆるインディーズゲームの波がやってきて、少ない予算、少ない人員であってもゲーム開発を行い、自分の作ったゲームを比較的自由に世の中に送り出せるようになってきました。これは、ゲーム開発者たちにとって朗報であり、ゲーム開発者を目指す人たちにとっても良い時代の到来となっています。

　本書では、こうした流れのなか、読者の皆さんが個人でもゲームを開発し、自らの手で作品を公開できるような方法を提案しています。この本を読むことで、実際にゲームをどのように企画するのか、どのようにプログラミングをするのかの概略をつかむことができるでしょう。ただし、本書で紹介するのは、シミュレーションゲームを題材にしたゲーム開発の一つの方法論にすぎません。今後、読者の皆さんがゲームを開発をしていくなかでは、自分なりの開発手段や方法を確立していく必要があります。この本はその手助けとなることを目指して作られました。本書を読むことで実際にゲーム開発をする意欲が湧いてくるとしたらこれほど嬉しいことはありません。

　また本書の著者であるロバート·ジェイ·ゴールドは、ゲームプログラマーとして複数の大手ゲームパブリッ

シャーにおけるゲーム開発を経験しており、現在はインディーズゲームメーカーであるGAMKIN株式会社のCTO（最高技術責任者）兼CGD（チーフゲームデザイナー）として活躍しています。彼も含めた我々GAMKINは、自分たちが培ってきたゲーム業界での経験をより多くの後進の方々と共有していくこと、また若手ゲーム開発者の育成の手助けとなることを考えています。私自身もゲーム専門学校で講師をしており、今回私たちの思いが一つの形となったことは、会社の代表として本当に嬉しく思っております。

最後になりますが、本書の刊行にあたり多大なるご尽力をいただきました株式会社翔泳社編集部の皆様、本書のイラストを担当してくれた、たえ☆ぽん様、またGAMKINのメンバー、GAMKINを日頃から応援・支援してくださっている多くのファンや企業の皆様、そして本書を手に取っていただいた皆様に対し、この場を借りて深く感謝の意を述べさせていただきます。本当にありがとうございます！

2014年11月吉日
沖永賢吾　代表取締役
GAMKIN株式会社

CONTENTS

Part1 シミュレーションゲームの基礎を知ろう

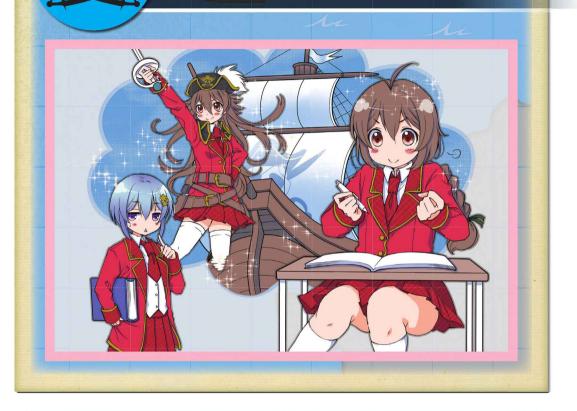
Part2 ゲームの基本機能を作る

Part3 エフェクトを追加しよう

Part4 コンピュータ対戦への道

サンプルコードダウンロードのご案内
本書の中で紹介したサンプルコードなどは、以下のサイトからダウンロードできます。
■翔泳社「サンプルダウンロード」のURL
http://www.shoeisha.co.jp/book/download/9784798137841

Part

1 シミュレーションゲームの基礎を知ろう

Chapter 01

シミュレーションゲームの歴史

本章で勉強すること

- シミュレーションゲームはコンピュータゲームの登場以前からあるゲームジャンル
- チェスや将棋の先祖となるゲームなど、古い歴史がある
- 近代の兵棋(へいぎ)演習から、ゲームに確率と資源という概念が導入される
- コンピュータとともに登場したゲームはシミュレーションだった！
- 多くのテーブルトップゲームから、コンピュータによるシミュレーションゲームが生み出された

シミュレーションゲームってどんなゲーム？

シミュレーションというと、「現実世界の現象を何か別の手段で置き換えて再現すること」をいいます。

したがって、**シミュレーションゲーム**といえば、現実世界の再現をゲームの形で表現したものといえるでしょう。

一般的に言うシミュレーションゲームは、ウォー・シミュレーションゲームのことを指すように思いますが、ウォー（戦争あるいは戦闘）以外にも、実際には「育成シミュレーション」「経営シミュレーション」「恋愛シミュレーション」などのいろいろな種類のシミュレーションゲームが存在します。これは（正確な再現ができているかはともかく）「何らかの現象を再現しようとするタイプのゲーム」ということになります。しかし、たんに「シミュレーションゲーム」というと、大抵の場合、ウォー・シミュレーションを思い浮かべることが多いのではないかと思います。

では、シミュレーションゲームの起源を探していきましょう。皆さんは、自分が生まれて初めて遊んだシミュレーションゲームを思い浮かべてしまうかもしれません。しかし、ビデオゲームが誕生する遥か昔からウォー・シミュレーションゲームは存在しており、深くその歴史を調べれば、その起源は数千年前に遡ることができます。その意味では、シミュレーションゲームは人類で一番古いゲームジャンルであるかもしれません。

チャトランガ、チェス、将棋

「戦争」「シミュレーション」「ゲーム」この3つの条件を満たす、古代インドのボードゲームとしてチャトランガ（Chaturanga）が知られています。王、将、象、馬、船、歩兵の駒を使うゲームで、4人制でも遊べるものでした。このゲームは３世紀（280年）頃から遊ばれていて、皆さんがよく知っている将棋やチェスの起源となっています。

図1-1 古代インドのボードゲーム「チャトランガ」

チャトランガから生まれた現代のチェスに近いルールのゲームは７世紀にはペルシア王国に存在していました。また、日本の将棋の元となるゲームは中国経由で日本にもたらされており、日本では「平安将棋」

が12世紀ごろに誕生しました。それぞれ16～17世紀ごろにはルールが標準化され、ゲームとして発展し始めました。

兵棋演習とウォー・ゲーム

チェスや将棋がシミュレーションゲームの祖先であることは間違いありませんが、これらは今みなさんがイメージしている現代のシミュレーションゲームよりも「アブストラクト」（抽象度）が高くなっています。現在のシミュレーションゲームのスタイルと雰囲気は、どちらかというと昔から軍人が利用している**兵棋演習**（へいぎ）や**作戦会議**に近いものでしょう。兵棋演習では、盤に戦場となる地形を描いて、人や馬、兵器などの駒を置いて作戦を話し合います。こちらはシミュレーションではあっても、ゲームではありません。しかし、このような兵棋演習は数千年前から存在します。チャトランガやチェス、将棋は、これらを抽象的化した結果生まれたゲームだといえるでしょう。

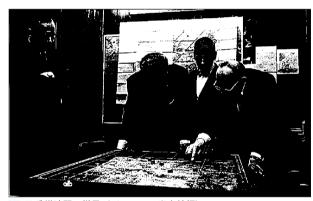



図1-2 兵棋演習の様子（ジョンソン米大統領）

兵棋演習は18世紀にひとつの重要な変革が起きます。1780年代にドイツのヨハン・ヘルヴィッヒはチェスのゲーム性からインスピレーションを得て、その後、軍人が使っていた兵棋演習とゲームを組み合わせたクリーグスピール（kriegsspiel）というものを作りました。これは、軍事訓練の道具と遊びを目的に、プロイセン王国の王様のために作られたものです。

このクリーグスピールにはチェスにも兵棋演習にもなかった要素が追加されました。その時代までの兵棋演習は動かし方などのルールを決めたチェスと同じようなものでしたが、確率の要素が加わったのです。クリーグスピールではサイコロの利用により、確率と乱数がゲームのシミュレーションの結果に影響するようになっています。

このおかげで兵棋演習に確率の概念が入ることとなり、いまみなさんがイメージしているシミュレーションゲームに近い道具立てと手法が実現するようになってきました。

図1-3 1913年のH. G. ウェルズによる「Wargaming」

テーブルトップのウォー・ゲーム

このような軍人の兵棋演習タイプのゲームをもとに、1970年代から1990年代前半にシミュレーションゲームのテーブルトップゲームやボードゲームのブームが起きました。盤の上に駒やときには小さな人形を置くタイプのゲームです。

最初は歴史的に有名な戦闘を再現するテーブルトップゲームから始まりましたが、まもなくSFや中世ファンタジーをモチーフにしたゲームも増えてきます。

このようなテーブルトップのゲームは、友達と協力したり競争したりすることができ、大会などのイベントもありました。その時代としては家庭用ゲーム機のシミュレーションゲームよりもルールが複雑であり、チャレンジ的要素が高く、ある意味でマルチプレイヤーにも対応しているというものでした。

(by Arnaud Ligny)

図1-4 イギリスのテーブルトップタイプのウォー・ゲーム「ウォーハンマー」

このテーブルトップウォー・ゲームの影響下から『ダンジョンズ・アンド・ドラゴンズ』(Dungeons & Dragons) という初のテーブルトークのRPG (Role Playing Game) が生まれました。これらは複数のプレイヤーがそれぞれパーティ (チーム) の一員の役割を担当し、ゲームのルールに従って行動や戦闘を行って冒険をするというスタイルでした。この『ダンジョンズ・アンド・ドラゴンズ』はコンピュータを使ったRPGゲームの起源となっています。『ウィザードリィ』、『ドラゴンクエスト』、『ウルティマ』や『ファイナルファンタジー』も、ダンジョン・アンド・ドラゴンズのシステムをコンピュータ化したものから発展してきているといえるでしょう。

コンピュータ・シミュレーションゲームのはじまり

ここまではアナログのシミュレーションゲームの歴史を追いかけましたが、コンピュータとシミュレーションゲームの関わりを見てみましょう。

1947年にアラン・チューリング (計算機科学および人工知能の父) が、単純化したチェスをコンピュータで遊べるようにしました。もちろん当時のコンピュータは建物ほども大きく、画面もなくて、大学や政府の研究機関でしか利用していなかった非常に貴重な装置でした。今のPC (パーソナルコンピュータ) とは姿や実態はだいぶ違いましたが、チューリングが作ったこのチェスのゲームは世界初のデジタルコンピュータゲームでした。

同じ1947年に、今度はトーマス・ゴールドスミス (アメリカのテレビ業界のパイオニア) がテレビの画面を利用した世界初のビデオゲームを開発しました。ただし、このゲームは今のコンピュータやデジタル回路を用いたビデオゲームとは大きく異なっており、電気科学的な仕組みで実現されていました。作られたのは、ピンボールゲームのような形をしたゲーム機です。そして、この世界初のテレビゲームのテーマはミサイルを撃つシミュレーションゲームでした! というわけで、コンピュータゲームもビデオゲームも1947年に開発され、その始まりは両方ともシミュレーションゲームだったわけです。

1970年代になるとメインフレームという強力なコンピュータが大学に導入され、コンピュータに興味のある学生が増えて、予約さえすれば誰でも使える時代になりました。これとテーブルトップゲームの影

図1-5 メインフレーム (IBM System/360)

響で、学生達はウォー・ゲームやテーブルトークRPGをコンピュータ上で再現できないか模索し始めました。

これにより複雑でモチーフ性の強いゲームが増えてきて、今のシミュレーションゲームのジャンルがほとんど固まってきます。当時のルール、仕組みやシステムは今の時代のシミュレーションゲームにも受け継がれています。

家庭用のシミュレーションゲーム

1980年代からようやくパソコンと家庭用ゲーム機が普及し始め、さまざまなゲームが増えてきました。そのなかでシミュレーションゲームももちろん一般の人に影響を与えるようになりました。

筆者の記憶のなかで一番古く遊んだ家庭用ゲーム機のシミュレーションゲームは1983年の『M.U.L.E.』（Ozark Softscape開発）でした。

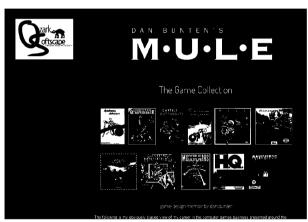



図1-6 Ozark Softscape社の現在のWebサイト

M.U.L.Eでは開拓惑星上で食料、エネルギー、鉱石、宝石を生産して、ゲームの中で経済圏を作ります。コンピュータが対戦相手になりますが、ほかのプレイヤーが操作することもでき、4人までのマルチプレイも実現していました（日本でもNECのPC-8801用に移植されて販売されていました）。

ちなみに日本の30代以上の人に初めてのシミュレーションのゲームを聞くと、1985年の『大戦略』シリーズ（システムソフト）、と1988年の『ファミコンウォーズ』（任天堂）のインパクトが強かったようです。

Chapter 02

シミュレーションゲームの種類

本章で勉強すること

- 世の中にはたくさんの種類のシミュレーションゲームがある
- まずシミュレーションゲームをスケール別に3種類に分類してみよう
- 戦闘シミュレーションとは異なる新しいゲームのタイプも紹介します
- シミュレーションゲーム内にあるさまざまな要素について解説します
- ターン制度、フェアプレイの概念、地形の影響、陣形や協力プレイなどの特徴について知識を深めよう。

シミュレーションゲームのコンセプトとその歴史を見たところで、今度はゲームシステムとしての分類を考えてみることにしましょう。たとえば、シミュレーションの対象によっても変わってきますし、新しいタイプのシミュレーションゲームも生まれてきています。こういったシミュレーション対象の違いはゲ

ームの仕組み（ルール）にも直接影響してきます。自分がどんなタイプのゲームを作ってみたいかよく考えて、既存のいろいろなゲームも参考にしていきましょう。

シミュレーションのスケール

第1章でも説明したとおり、シミュレーションの対象にはさまざまなものが考えられますが、今回は典型的なウォー・シミュレーションゲームを扱いますので、これらについてすこし詳しく見てみましょう。ウォー・シミュレーションゲームの場合、戦争（あるいは戦闘）をゲームのシステムやルールでどう表現するかが主眼となります。

戦争から戦闘をシミュレーションするゲームでは、基本的に戦いのスケール（規模）でゲームの仕組みやユニットが変わってきます。そのためゲームのスタイルを大きく3つの種類に分けてみることにしましょう。ここでは「戦争スケール」「作戦スケール」「戦術スケール」に分けてシミュレーションゲームを考えてみます。

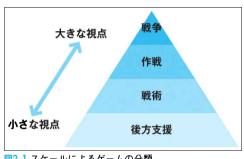



図2-1 スケールによるゲームの分類

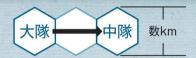
戦争スケール

ターン：数日から数週間
マス：数十キロメートル以上
ユニットの単位：数百～数千人

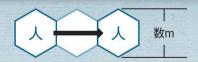
このスケールのゲームではプレイヤーはひとつの戦いではなく、戦争という出来事をシミュレーションすることになります。プレイヤーは大まかに国を単位としてユニットをコントロールをしながら相手国と戦います。ターンは数日から数週間くらい。1マスは数十キロメートル程度になります。ユニットは数百から数千人単位の師団や軍団のゲームが多いでしょう。そして移動と戦いは基本的に隣のマスに限ります。この分野のボードゲームは多いですが、ビデオゲームでこの規模のゲームは最近では少なくなっています。

『エイジオブエンパイア』シリーズ（http://www.ageofempires.com/）がこのスケールのシミュレーションの代表だと考えられます。

作戦スケール

ターン：数時間から数日
マス：数キロメートル以上
ユニットの単位：数十～数百人

作戦級のゲームではプレイヤーはひとつの大隊や中隊くらいの単位を受け持つことになります。ゲームが対象とするのは戦略スケールにおける戦闘くらいの範囲です。このスケールのゲームでは戦闘をする単位以外にリソース（食料や武器などの資源）の管理、建物や研究などのシミュレーションも含まれています。このスケールのゲームのタイトルは比較的多くありましたが、最近はほとんどがシリーズゲームになってきているようです。このスケールの代表的なタイトルとしては『スタークラフト』シリーズ（http://us.blizzard.com/en-us/games/）や『信長の野望』シリーズ（http://www.gamecity.ne.jp/）などが考えられます。

戦術スケール

ターン：数秒から数分
マス：数メートル
ユニットの単位：1人、1機

戦術スケールのゲームではプレイヤーは数ユニットのひとつの小さな戦いのシミュレーションをしています。1ターンは数秒から、1つのマスは数メートル、1つのユニットは1キャラクター程度です。このスケールのゲームはキャラクターにフォーカスをするのでRPG的な要素を持つこともあります。戦術は英語で言うとTacticsです、したがって『ファイナルファンタジータクティクス』（Final Fantasy Tactics）などタイトルにTacticsが入っているゲームは基本的にこのスケールに当てはまるでしょう。

シミュレーションゲームの新しいジャンル

上の3つのスケール以外にもいくつかシミュレーションのゲームのジャンルが存在します。近年流行のサブジャンルとしてタワーディフェンス系とMOBA系を紹介しておきます。

タワーディフェンス

図2-2『Plants vs. Zombies』

タワーディフェンスは防御に注目したゲームです。ゲームの勝利条件はほかのシミュレーションゲームとだいぶ異なり、相手を全滅させるのではなく、自分が最後まで生き残ることです。このジャンルのゲームでは、プレイヤーは動かないユニットを配置することで自分の基地を敵の大軍から守ります。ユニットの配置、それらの成長と全体的なリソース管理が非常に大切となります。このスタイルは昔からあり、もともとマイナーなジャンルではありましたが最近では『Plants vs. Zombies』シリーズ（http://www.popcap.com/plants-vs-zombies-1）のお陰で注目されるようになってきました。

MOBA

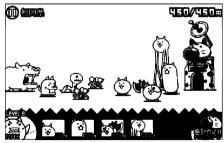



図2-3『にゃんこ大戦争』

MOBA (Multiplayer Online Battle Arena) 系のゲームはある意味タワーディフェンス系の真逆です。MOBAではプレイヤーは自分のリソースと研究（アップグレード）で自分のユニットを短時間でたくさ

ん作り、敵軍と戦わせて基地を破壊することが目標になっています。防御はほとんど関係なく、攻撃することが重視されています。このゲームのジャンルは『ウォークラフト』(http://us.battle.net/wow/en/game/guide/)のプレイヤー達がそのゲームのマップエディタを利用して作ったゲームシステムが元になっています。今はこのゲームシステムを元にカジュアルゲームがたくさん登場してきています。2012年に登場して人気ゲームになった『にゃんこ大戦争』(http://www.ponos.co.jp/iphone/jp/appli/battlecats/battlecats.html)はそのひとつといえるでしょう。

シミュレーションゲームのゲームシステム

シミュレーションゲームはたくさんあって､それぞれにいろいろな機能とゲームメカニズムがあります。でもシミュレーションを代表する典型的なシステムもあります。もちろんこれらは必須ではなく、これらの有無がシミュレーションゲームであることを決めているわけでもないのですが、シミュレーションゲームで遊んだりゲームデザインしたりするときには、こうした特徴を意識するとよいと思います。

ターン制とリアルタイム制

これはゲーム中の時間の流れをどのように管理するかという方法論です。TBS(Turn-Based Strategy)はターン制による管理方法でプレイヤー(あるいはコンピュータが)が順に行動していくというものです。一方、RTS(Real-Time Strategy)はゲームの中の時間の流れが決まっており、その流れの中で行動します。ターン制のほうが次の行動をゆっくり考える時間がありますが、リアルタイム制の場合は同じ行動をしてもタイミングによって結果が異なってきたり、行動力や決断力がゲームの一要素になってきます。

フェアプレイ

シミュレーションゲームでは自分と相手のユニットは基本的に同じくらいの強さや性能を持っています。その理由はお互いがフェアプレイをしていると戦略が重要要素となり、より高い戦略を持つプレイヤーが勝つことになるからです。逆に物語要素の強いRPG的なゲームでは自分と相手の性能に違いがあります。たとえば『ファイナルファンタジー』シリーズでは、プレイヤーキャラと敵キャラのHPとダメージに桁が異なるほどの差がついていました。

リソース管理

ゲーム中にはお金や資源などのリソースがあり、それらを増やせるようになっています。リソースはユニットをアップグレードするための研究費用として、あるいは新ユニットの生産で利用できます。この仕組みによってプレイヤーはゲーム中に成長していくことになります。

ユニットの視界

プレイヤーが見える物とユニットに見える物は異なります。プレイヤーはゲーム全体を見渡すことができますが、各ユニットは行動範囲が限定されます。たとえば、プレイヤーからは敵の位置が見えていても、ユニットの攻撃範囲の外にある場合は攻撃できません。

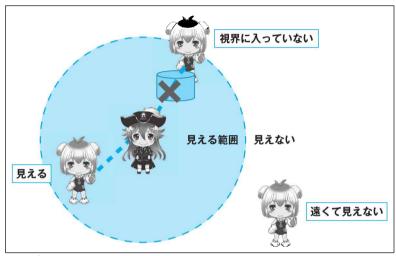



図2-4 プレイヤーの見えている範囲とユニットに見えている範囲は異なる

地形の影響

シミュレーションゲームのフィールド（マップ／面）にはいろいろな地形のタイプ、高さ、障害物などが表現されます。こうした地形を戦略的に利用することが可能です。地形を活用することで状況が有利にも不利にも働きます。たとえば、高い所から低い所を攻撃すると攻撃力が上がる、または平らな草原より森で移動するほうが移動力を消費するなどです。

ZOC（Zone of Control）

それぞれのユニットは自分の周りのマスに影響を与えられます。この影響範囲をZOC（Zone of Control）と呼びます。たとえば、ZOC内では敵ユニットが自由に移動できない、逃げると自動的に攻撃される、またはZOCには入れるが通過ができないなどのルールを設けている場合があります。

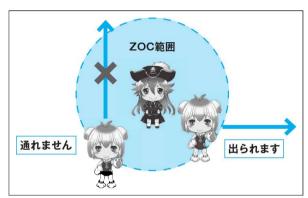



図2-5 Zone of Controlの概念

ユニットの向き

ユニットには、正面、側面、背面などが表現されていることがあります。たとえば、背面から攻撃をすると正面からそのユニットを攻撃するよりも防御力が下がるなどです。

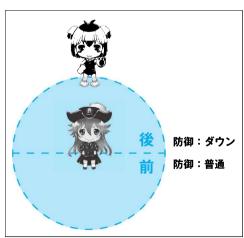



図2-6 背面から攻撃

陣形などの影響

複数のユニットが同じ敵を囲むとその敵の防御力は下がります。

図2-7 挟んで連係攻撃

協力ボーナス

ユニットの近くの味方から影響が与えられる、または味方に対して影響を与えることができます。たとえば、リーダー級のユニットは周りのユニットを強くする、または仲良しのユニットが近いと両方が盛り上がることで戦闘力がアップするなどです。

図2-8 協力ボーナス

以上、ここではシミュレーションのゲームジャンルやルールについて考察してきました。これらを踏まえて、次の章ではこれからこの本で作るシミュレーションゲームのデザインを考えてみることにしましょう。

Chapter

03 ゲーム企画書(GDD)を書いてみよう

本章で勉強すること

- GDDはゲームコンセプトと開発のための設計図
- プレイヤーのためにではなく開発をする人のために作ろう
- GDDは変化する。開発の各フェーズで順次GDDを修正していこう
- イラストや図を工夫してイメージしやすいものにしよう
- 典型的なGDD、完全なGDDというものはない。開発チームの個性に応じた自分なりのGDDを作ろう

プログラマー、アーティスト、ゲームデザイナー各１名、またはそれ以下の規模（最近だとインディーズと呼ばれるゲーム開発スタイル）だと、しっかりとしたゲームの企画書を用意する必要はあまりありま

せん。逆に、小規模なのにやたらに企画書に凝りすぎると、いつまでたっても完成しない失敗企画の元になったりします。もちろん、規模が大きくなってくれば、企画書の重要性は増してきます。

小規模のチームであれば、自分たちが何を作りたいかは簡単に共有できますし、途中で大幅な変更があっても、それぞれの役割を理解して、問題を解決していけます。しかし、最低限の方針を決めたり、メンバーで共有するメモや手引書、参考資料などは準備したほうがよいでしょう。

今回、みなさんと一緒に作っていくゲームは小規模な開発を想定しているので、完璧な企画書は必要ないかもしれません。しかし、本書では読者の皆さんが将来的にゲーム開発プロジェクトを推進することをイメージして、一連の流れを説明してみようと思います。また、この本で開発するゲームに関して、読者の皆さんとの共通認識が得られるように、開発内容を決めておきたいと思います。

アイデアを形に、ゲーム企画書（GDD）とは

では、まずゲームの企画書（GDD: Game Design Document）とはどんなものでしょうか？Wikipediaでは次のような説明があります。

ゲームデザインドキュメント（GDD）とは、ビデオゲームに関する設計について絶えず更新され、または編集されていく企画書のことである。GDDは、開発チームによって作成され、編集されるものであり、そしてそれは開発チーム内の取り組みを整理するために主としてゲーム開発企業で使用されるものである。そのドキュメントは、ゲーム開発が進行されるあいだずっと使用される指導的な役割を果たすビジョンとして、チームのゲームデザイナー、アーティスト、プログラマーとのコラボレーションにより生み出される。-wikipedia（英語ページから訳出）

この説明をもとに、ここではGDDについて3つのポイントを挙げておきましょう。

- **ゲーム開発者達によって作成されるドキュメント（資料）であること**
- **GDDの対象となる読者は、各分野のゲーム開発者達であること**
- **絶えず更新され、または編集されていくドキュメントであること**

それぞれについてポイントを見ていくことにします。

ゲーム開発者達によって作成されるドキュメントであること

GDDは通常、開発経験のあるゲームデザイナー、アーティスト、プログラマー達が集まった開発チームメンバーが、自分たちで使う目的で作成します。開発チームが一人の場合は、こういったものを一人で

作成しなければならず、未経験者にとっては大変かもしれませんが、できるかぎりの想像力を働かせて作成するとよいでしょう。

GDDの対象となる読者は、各分野のゲーム開発者達であること

GDDを制作した経験があまりないと、プレイヤー（ユーザー）目線でGDDを作ってしまう傾向があります。しかし、GDDの読者はゲーム開発者であるため、開発者達がそのGDDを見て開発内容が分かるようにします。よくできたGDDには、ターゲット、ゲームプレイ、ゲームアート、レベルデザイン、ストーリー、キャラクターやビジネスのことが簡潔かつ明確に纏められているものです。

絶えず更新され、または編集されていくドキュメントであること

ゲームの開発には複数の開発段階があります。また、最近のゲームは、プレイヤーの動向に応じて、絶えずアップデートや改善が行われていきます。このような開発フェーズに合わせて、GDDはしばしば変更され、改編され、拡張されていくことになります。当初のGDDは、いくつかのコンセプトとアウトラインから始まり、次第に完成形になっていきます。各開発工程からプロジェクトが終了するまで、常に指針の役割を果たしていくことになるのです。

ゲーム開発の3つ（+1）のフェーズ

さきほど「ゲームの開発には複数の開発段階がある」と説明しましたが、商業向きのゲームの開発であれば大きく3つの開発フェーズと運用フェーズがあります。この本では一人（あるいは数人）で開発していくことが前提ですが、開発工程についても一応頭に入れておくと便利です。

コンセプトワーク

GDDは、まずコンセプトドキュメントという形で始まります。ここで一番重要なことは、この時点での企画のセールスポイント（達成すべき方針と目標）が記載されており、開発者達にとってどのくらい魅力的か（つまり、このゲームがいかに他人に興味を持ってもらえるか）です。

プリプロダクション

次のフェーズでは、コンセプトに沿って、実際にゲームを簡単に製作してみます（ゲームのタイプによってはペーパープロトタイプでも可）。製作をしていくなかで、開発チームのメンバーと活発な意見交換をして、出てきたさまざまなアイデアをメモします。これらをまとめていくことによって、どのアイデアがこのゲームにとって適切なものであるのかを判断し、取捨選択します。この段階で重要なのは、このゲームの本質的なアイデアを見抜き、それをブラッシュアップしていくことです。

製作、プロダクション

製作フェーズに進むと、ゲームのコアとなる部分は固まり、簡単に内容変更ができない状況になっています。もちろん開発チームの了承のうえで変更しても問題ありませんが、影響は大きくなります。このフェーズでは、実際にリリースするゲームの開発が進んでいるので、出来上がっていくゲームについてデバックという作業も発生します。

（↗）

運用

実際にゲームをリリースしたあとは、運用というフェーズになります。このフェーズでは、今までGDDに記載できなかった（優先できなかった）アイデア、プレイヤーからの意見の取り込み、あるいは開発途中に思いついたアイデアが実現できるようになります。製作フェーズでGDDに記載できなかった場合でもあきらめる必要はまったくありません。ゲームが安定的に稼動したときに、再度アイデアが実現できるか、開発メンバーに相談してみるとよいでしょう。ただし、ゲームの面白さの向上に合うものであるかという点を常に自問自答してほしく思います。

これから作るゲームの話

さて、ここからは実際に私たちが作るゲームについて考えていきましょう。本当は読者の皆さんに会って色々と話し合いながらゲームの題材など決めていけるとベストですが、紙面の都合上それは難しいので、こちらから色々と提案する形で進めていきます。

まずは概要からです。ゲームのジャンルは、シミュレーションゲーム（もちろん！）、その中でも前章で話したタクティクス分野のゲームを作ることにします。また中世風の騎士物語的なモチーフはありがちなので、今回は海賊を題材とします。あとはマーケティング予算などはありませんので、できるだけ多くの人に遊んでもらうことを前提に、Webブラウザで動くゲームを作りましょう。

次に、ゲームの内容に関してコンセプトを出しあって軽くブレインストーミング（ブレスト）をしましょう。

モチーフは海賊

- 海賊船
- 海や島を探索する
- ほかの海賊船と戦う、宝の奪い合い
- ロマンあるストーリー
- 大砲などの武器
- 港でモノを買ったり、売ったりできる
- 海賊にはキャプテンが存在
- 新しい海賊仲間を増やしていく

アイデアとゲーム企画との関係

企画を考える際に、ちょっと注意しておきたいことがあります。それは、アイデアを出すこと自体にそれほど重要性はないということです。
もし、あなたがじっくりとゲームの特徴を考えた場合、いろいろなアイデアが出てくるはずです。しかし、大切なのはそのアイデアがゲームとして使えるかどうか、開発に繋がるアイデアなのかどうかです。現在の開発の体制（開発期間／知識／プラットフォーム）でどのように実現できるかを考え、最終的な取捨選択の判断をします。取捨選択を通過して初めて、ゲームの企画に繋がるアイデアといえるということです。
初期のコンセプトの段階でアイデアを出すことは必要かもしれませんが、最終的にゲーム開発に繋がるアイデアに落とし込むことが重要です。

ちょっと考えるといろいろとな面白いアイデアが湧いてきます。しかし、今回は読者のみなさんの余暇の時間でゲームを作るので、時間と開発能力、その他を勘案していったん、以下のような厳しい制約を設けましょう。面白いアイデアは最初の開発が終ったあとに、運用フェーズで追加することをお勧めします。

- **ブラウザで動くゲームにする**
- **1回のゲームプレイ時間は5分前後**
- **開発期間は1か月以内**
- **キャラ数は、コスト面を考えてあまり増やさない**
- **本格的なストーリーやRPG的な要素は、時間がかかるので今回は採用しない**

最初に説明したとおり、これからGDDは変化していきますが、まずは初期のコンセプトワーク段階でのGDDを確認してみます。
GDDを書くツールは自由ですが絵などがあったほうがよいのでグラフィックを扱う機能があったほうがよいでしょう。ここではほかの関係者にプレゼンすることも前提にMicrosoft PowerPointを使って作っていきます。

GDD①～ゲームの概要～

PIRATE TACTICS
パイレーツタクティクス
BATTLES AT HIGH SEA
外海の戦い

ゲーム概要
パイレーツ・タクティクスは海賊が戦うカジュアルなシミュレーションゲームです。

バックグラウンド
パイレーツタクティクスは、海賊のロマンをベースに開発するオリジナルIP（独自の知的財産）です。

ターゲット
客層：タクティクスやシミュレーションが好きなプレイヤー
場面：短時間に気軽に遊べる、カジュアルゲーム
プラットフォーム：ブラウザゲーム（サーバー無し）

キー・フィーチャー
•カジュアルタクティカルシミュレーション
•スキルを利用して勝利を手に入れよう！
•Twitterでプレイの結果をシェア出来ます！

図3-1 初期段階のGDD①（ゲームの概要）

タイトルとサブタイトルについて

ゲームタイトルは、ゲームコンテンツと一致するもので、しっかりと意味があり、記憶されやすく、思い出されやすいものであるほうがよいでしょう。また、第三者によって使用されていない（商標登録等がされていない）必要があります。サブタイトルを加える場合もあります。サブタイトルを加えることにより、ゲームの様子がより伝わりやすくなります。

ゲーム概要（エレベーターピッチ）

最初にゲームの概要を記載します。そこでは、このゲームの特徴（どのようなゲームであるのか）を簡単に説明していきます。詳細については、以降の各パートで明らかになりますので、ここではゲームコンセプトについて最も重要なことを説明していきます。

エレベーターピッチ（Elevator Pitch）とは、「迅速かつ明確に短い時間で相手に価値ある情報を伝えられるか」という意味で使われます。その内容は、ケースバイケースですが、口頭なら時間のイメージは30秒以下です。なぜ「エレベーター」かというと、エレベーターで出会った相手に対して、そのちょっとした時間を活用して相手に興味を持ってもらえるような説明ができる、という考えから来ています。あまり独創的な方法ではありませんが、誰でも分かるような有名なタイトル（ゲーム、映画など）を利用して例えるという方法もあります。

03

バックグラウンド

ここでは、どのような背景でこの企画が行われたか、またどのような環境で開発を進めるのかを説明します。たとえば、オリジナルIP（知的財産）のゲームなのか、有名なIPを利用するのか、このゲームの企画に影響を与えたゲームコンセプトの有無を説明しておくとよいでしょう。また、開発するツールやゲームエンジンなどが決まっている場合は、その情報も説明しておきましょう。

ターゲットオーディエンス&プラットフォーム

ゲームの対象者は誰なのか、どのような人が興味を持ってくれるのか、それらの人々はどんな属性を持つのかを想定していきます。たとえば、多くのコンシューマーゲームは、プレイヤーに長時間のプレイを要求します。そのかわり、プレイヤーはゲームのプレイを通じてより深みのある体験ができます。一方、カジュアルゲーム（モバイルソーシャルゲームやPCブラウザゲームなど）は、プレイヤーに対して長時間の拘束を要求しません。ほかの活動の合間にも気軽にゲームプレイが可能となります。また、プレイヤーにプレイに関する記憶を要求するものは少なく、短時間でゲームを楽しめるようになっています。

チームの能力と労力も考えて、どのプラットフォームを採用するかも考えましょう。十分な開発経験もないままコンシューマーゲームを作るのは現実的ではありません。そうであれば、Webやモバイル向けのゲームを開発したほうが安全かもしれません。また、ゲームがマルチプレイもしくはソーシャルネットワークと関係しているのであれば、それも記載しておきましょう。

キー・フィーチャー

ゲームの重要な要素（特徴）をまとめていくセクションです。とくに既存のゲームよりも優れている点をまとめ、目指すべきゴールも定めていきます。項目については、箇条書きにしていけばよいでしょう。ここで纏められた内容はゲームのPRにも使えますので、ゲームのウリとなるものをしっかり纏めましょう。

GDD②～ゲームの世界観～

ゲームの世界観

ストーリー

海賊時代。各王国所属の海賊たちは、未開の地を探索し、その地に存在する宝を奪いあっている。

ビジュアルスタイル

海賊たちは、全員女性で格好可愛い系。ゲームの戦闘シーン上では2頭身にデフォルメされる。

オーディオスタイル

ゲームの曲（BGM）は、海賊をイメージしたファンタジー音楽にします。効果音（SE）は、シンプルにします。大砲の撃つ音や船の音などに限られます。各海賊キャラには、声は付けません（声優は起用しません）。

図3-2 初期段階のGDD②（世界観）

ストーリー

必要ならば、物語や時代背景等の設定に関する簡単な説明を入れておくとよいでしょう。その際、その設定がゲームにどのような影響を与えるのかを明確にしておきましょう。たとえば、海賊時代をゲームの基本設定とした場合には、ゲームプレイとして軍艦での戦闘と財宝略奪が含まれることを説明する必要があるかもしれません。

ビジュアルスタイル

このセクションでは、ゲームの概観や雰囲気をビジュアル的なコンセプト（コンセプトアート）で説明していきます。その際に制作されるイラストはゲーム内の設定に応じて、現実世界 or ファンタジー世界、過去 or 未来、宇宙 or 地球、もちろん2D or 3D等を意識したものである必要があります。コンセプトアートはできるだけ入れたほうがよいでしょう。

オーディオスタイル

　ここでは、ゲームにおける音声がどのようなものになるべきかをまとめます。ゲームの世界観や設定のイメージを膨らませ、どのような曲調やスタイル（効果音等も含む）がこのゲームにとってよいのかを考えていきます。場合によっては、プロの音楽家や声優に依頼したいという案も出てくるでしょう。初期の段階ではそれらをリスト化し、自分たちのゲームにとってベストな形でまとめて、最終的に絞っていくということでも大丈夫です。

　インディーズの開発では、予算も人手も足りないので、フリーの音素材等を探してゲームに使用することもひとつの手でしょう。フリー素材であっても許諾が必要な場合もありますので、著作者に正式な手続きを取って使用するようにしてください。

GDD③～ゲームキャラクター～

ゲームのキャラクター
プレイヤーキャラ
キャプテン
素早いキャラ
攻撃キャラ
防御キャラ
敵キャラ
ボスキャラ
手下キャラ
素早いキャラ
攻撃キャラ
防御キャラ

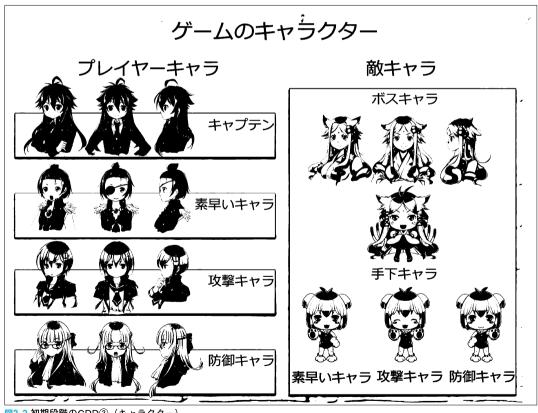



図3-3 初期段階のGDD③（キャラクター）

キャラクター（Characters）

　ここでは、ゲーム内のキャラクターや敵に関わる情報が主となります。たとえばRPGの場合であれば、主人公もしくはボス（敵）のビジュアルやちょっとしたシナリオを記載しておくことで、そのキャラクターがゲームの世界観や設定の中でどのような存在なのかが分かります。また、ジャンルによって差はありますが、プレイヤーがキャラクターを通じてどのような体験をするのかも意識して、どのようなキャラクターにすべきか考えていくとよいでしょう。

GDD④～インターフェイス～

ゲームのマップ

四角いマスを採用します（9x13）

キャラのユニット

マップ用ユニット

地形タイプ

通常海

浅い海

荒い海

ゲームマップのイメージ

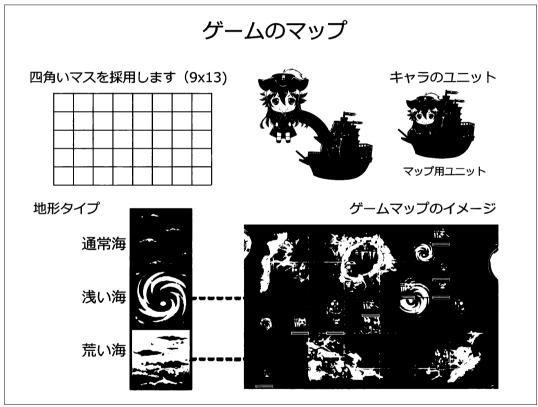



図3-4 初期段階のGDD④（インターフェイス）

レベルデザイン及び環境設計

　実際にプレイヤーがプレイする舞台（マップ等）はどのようなものか、またその舞台にはどのようなものが存在し、配置されているかを考え、実際に必要なコンテンツの量を定める必要があります。

レベルデザイン

ゲーム業界の中には、レベルデザインという単語を「難易度の調整」という意味に誤解している人がいます。この「レベル」という単語は日本語の「面」というゲーム用語に相当する言葉です。ステージやマップデザインに置き換えると分かりやすいですが、誤解のないようにしたいところです。レベルデザインの検討では、次の点を意識する必要があります。

- **各面を遊ぶことが楽しい（プレイヤーが感情移入できるようにする）**
- **遊びに自由を与える（選択に意味を与える）**
- **テンポ（流れ／リズム）が良い**
- **ゲームシステムを引き立たせる**
- **ゲームバランスを壊さない**

レベルといっても
難易度のことではありません

この章ではGDDの作成を見てきました。GDDはゲームの開発の各フェーズに適した形で作っていくことが重要と覚えておきましょう。その他、GDDを作成するときに注意すべき点をまとめておきます。

- **見やすいフォントの使用**
- **レイアウトの統一（情報の整理の仕方等）**
- **できるだけ短文を使用して、読みやすくする**
- **曖昧な表現は避け、具体的な表現にする**
- **複雑なものは見やすくする工夫をする**
- **イラスト（挿絵）をできる限り入れる**

最後に補足しておきたいことがあります。読者の皆さんはこの章を読んでGDDのアウトラインがある程度イメージできたかもしれません。しかし、すべてのゲームジャンルや条件を満たすような定型的なGDDアウトラインは存在しないということも覚えておきましょう。

たとえば、ドラゴンクエスト、マリオブラザーズ、コールオブデューティのようなゲームを考えた場合、それぞれが異なるGDDを有しています。さらに、仮にそれらのゲームのGDDを読者の皆さんが考えた場合も原作のGDDとは異なるはずです。

仮に今、ドラゴンクエストのGDDが目の前にあったとしても、自分のゲームプロジェクトにとっては、

意味のないセクションも多く含まれているはずです（もちろん、参考となる部分や必要なセクションが潜んでいることもあるでしょう）。大切なのは、オールマイティで定型的なGDDというものはなく、それぞれのプロジェクトごと、またチームによってもGDD作成方法は異なっていてよいということです。

Chapter 04 ゲームの開発環境

本章で勉強すること

- Webページは HTML/CSS/JavaScript の3つで作ることができる
- ブラウザ上のプログラムは JavaScript で書くことができる
- ブラウザには JavaScript のバグを発見するデバッガなどがついていて便利です
- ブラウザごとの違いを吸収するために「フレームワーク」を利用します
- enchant.js はゲームのための便利な機能満載のフレームワーク
- 本書ではブラウザは Google Chrome を用いることとし、テキストエディタは Sublime text を紹介します
- 使い慣れたエディタがあれば、そちらを使って進めよう

前章で決まったように、これから皆さんとブラウザでプレイする2Dのシミュレーションゲームを作っていきます。そこで、プログラミングを始める前にブラウザとそれに関する技術の知識をつけましょう。

Webページの表示の仕組み

ブラウザはWebページなどを表示できるアプリケーションソフトウェアです。おもにHTTP（Hyper Text Transfer Protocol）というプロトコル（通信の手順等を定めた規約）でインターネットからソースを読み込み、表示させます。単純にページを表示するだけでなく、いろいろなプログラムの画面を表示させて検索やショッピング、FacebookなどのSNS（Social Networking Service）そしてゲームなどもできます。

Webページを定義するおもな技術は3つあります。それはHTML、JavaScript、CSSです。

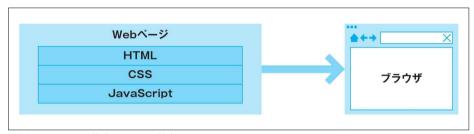



図4-1 Webページを構成するための技術

この３の技術はそれぞれ違う目的を持って使われています。

HTML

HTML（Hyper Text Markup Language）の役割はWebページの内容（文章など）を定義することとその配置です。HTMLの最新版はバージョン5（HTML5）となっています。前と後ろに不等号を使った<html>などのタグを使って内容を意味付けします。

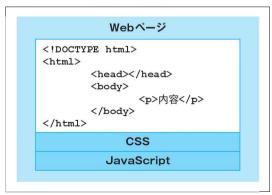



図4-2 HTML

HTML5はいままでのHTMLよりも高度な表現が可能となり、ゲームなどを作るのにも便利になりました。Canvasという画像や映像を高速に扱う新しい仕組みが追加されており、ゲームの画面を作るのにとても便利になっています。

CSS

CSS（Cascading Style Sheet）の役割はWebページのデザイン（見栄え）を定義することです。たとえばWebページのタイトルの文字の色やテキストのフォーマット、色や背景などはCSSで定義をします。

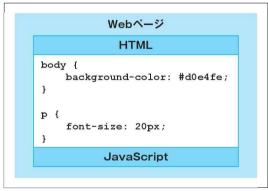



図4-3 CSS

今回作るゲームでは私達が直接CSSを利用することはありません。CSSについて詳しく知りたい場合は、CSSを解説した入門書やWebサイトもあるので、そちらを参考にしてください。

JavaScript

JavaScriptの役割はWebページの挙動です。

JavaScriptはプログラミング言語のひとつです。プログラミング言語というと、ほかにC言語、C++、Java、Rubyなどたくさんありますが、ブラウザの内部でプログラムを動かそうとする場合は多くのブラウザで採用されているJavaScriptを使うことになります。

▼ JavaScriptの例

```
<script type="text/javascript">
<!--
myTbl      = new Array("日","月","火","水","木","金","土");
myD        = new Date();
```

```
myYear     = myD.getFullYear();
myMonth    = myD.getMonth() + 1;
myDate     = myD.getDate();
myDay      = myD.getDay();
myHours    = myD.getHours();
myMinutes = myD.getMinutes();
mySeconds = myD.getSeconds();

myMess1    = myYear + "年" + myMonth + "月" + myDate + "日";
myMess2    = myTbl[myDay] + "曜日";
myMess3    = myHours + "時" + myMinutes + "分" + mySeconds + "秒";
myMess     = myMess1 + " " + myMess2 + " " + myMess3;
document.write( myMess );
// -->
</script>
```

(http://www.red.oit-net.jp/tatsuya/java/getdate.htmより引用)

本書で開発するゲームもそのほとんどはJavaScriptで書くことになります（JavaScriptについては次章で基本的な部分を説明します）。

どのブラウザでもこの3つの技術はほとんど同じように使えますが、細かい部分では例外もあります。とくにJavaScriptとCSSに関してはブラウザの影響を受けることが多く、パフォーマンスや仕様に微妙な違いが見られます。

技術面でのブラウザゲームのメリット／デメリット

どの技術にもメリットとデメリットがあるものです。デメリットのない万能の技術というのはまずありえないでしょう。しかし、自分の環境のメリットとデメリットを知れば、状況がコントロールしやすく、前に進むために必要なことも見えてきます。

まず、ブラウザでゲーム開発するときのメリットとデメリットをまとめてみました。

メリット：

・開発環境を無料で整備しやすい
・ブラウザゲームならどのOSやスマホでもある程度遊べる
・そのため、プレイヤーのリーチは広い
・ブラウザの技術はゲーム以外にも応用できる
・ブラウザの技術は基本的にすべてオープンで自由に使える

デメリット：
- **ネイティブのゲーム（OS上で直接動作するゲーム）より実行速度が遅い**
- **ブラウザの種類が多く、仕様の違いも思った以上に多い**
- **JavaScriptは非常に自由な（厳密ではない）言語なので比較的バグやミスが起きやすい**

以降ではこうしたデメリットにどう対応していくか考えてみましょう。

ブラウザゲームのデメリット対策

実行速度

ブラウザ上で動くゲームの場合、OSの上で直接動作するネイティブアプリケーションのゲームより実行速度は遅くなります。しかし、3Dや物理演算に頼るゲームでは実行速度は致命的な問題となりますが、これから作るような動きの激しくない2Dのゲームなら影響はそこまで大きくはないでしょう。

ブラウザなどへの依存

実際にブラウザはたくさん存在しており、環境依存も思ったよりあります。これはブラウザでゲームをゼロから作るときの一番の壁です。今なら主要なものだけでもこんなにたくさんのブラウザに対応しなければなりません。

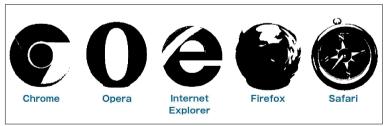



図4-4 いろいろなブラウザに対応しなければならない

各ブラウザごとに別々のプログラムを書いていては大変ですから、違いを吸収してくれる何かが必要です。それは**フレームワーク**または**SDK**（Software Development Kit）と呼ばれています。

イメージとしては、各ブラウザの開発の基盤は凸凹になっている感じです。フレームワークはプログラムとブラウザのあいだで動作します。フレームワークを使ってプログラムを書く場合、基本的に命令はフレームワークに対して出します。するとフレームワークは各ブラウザに合った命令を必要に応じで発行してくれるのです。

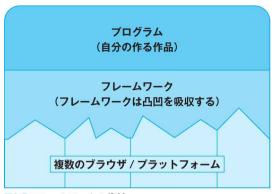



図4-5 フレームワークの役割

今回はゲームを作るので、ゲームに特化したブラウザのフレームワークで環境の違いを吸収します。今回利用するのは**enchant.js**というフレームワークです。enchant.jsにはゲームを作るために便利な機能がたくさん用意されています。

バグやミスが起きやすい

これも大きな問題ですが、JavaScriptに限らずどのプログラム言語でもバグやミスは付き物です。JavaScriptの場合、ほかの言語に比べて記述があいまいになりがちである精密な数字の演算に向かないなどの問題点もありますが、セキュリティはしっかりしていると筆者は考えています。プログラムの影響範囲もおもにブラウザに限定されているので、C++、C＃やJavaに比べ、OSのデータを破壊してしまうようなバグは作れません。また、バグの修正についても、昔は目と手でプログラムを確かめていましたが､最近のブラウザにはデバッガ（バグ発見の手助けをしてくれるツール）なども用意されていますので、ほかの言語との差はそこまでないかもしれません。

プログラムを始めるための準備

今回はGoogleのChromeブラウザで開発を行います。ChromeはどのOSのバージョンでもちゃんとした開発ツールになります。Internet ExplorerやSafariにも優秀な開発ツールはありますが、MacintoshやWindowsでの操作は異なるので統一されているものを採用しました。

Google Chrome - ブラウザ

図4-6 Google Chrome

Chromeの強みは、HTMLやCSSといったWebの新しい技術や規格が積極的に取り入れられていることです。そのためWebページやサービスの開発側から見ると、Chromeは対応しやすいブラウザです。

もし、Chromeを自分の環境にインストールしていない場合は、ネットからダウンロードして、インストールしておいてください。

Google ChromeのWebサイト
http://www.google.co.jp/chrome

SublimeText - テキストエディタ

ゲームやプログラム全般を書くために**テキストエディタ**というツールが必要です。テキストエディタはWordなどのような文書を作成するためのアプリケーションですが、文章の見た目を変更したりする機能はなく（基本的に文字サイズやフォントに変化をつけたりする機能はありません）、文書入力に特化した軽快さとプログラミングにも便利な機能を備えています。プログラミングには高機能な統合開発環境を用いることもありますが、JavaScriptの場合はテキストエディタで書くことが多いでしょう。

どのOSにも基本的に何かのテキストエディタが付属していますが、プログラミングには不便なことも多いので、使いやすいテキストエディタを準備しておく必要があります。

もし自分が使い慣れているテキストエディタがあれば、ぜひそちらを利用しましょう。「テキストエディタなんて使ったことないよ」という場合は、MacintoshでもWindowsやLinuxでも使えるSublime Textエディタをお勧めしておきます。このエディタは有料ではありますが、評価目的で利用することも可能です。

Sublime Textのインストールは簡単です。次のサイトから最新のSublime Textをダウンロードしましょう。

Sublime TextのWebサイト
http://www.sublimetext.com

図4-7 Sublime Textのサイト（http://www.sublimetext.com/）

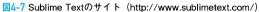
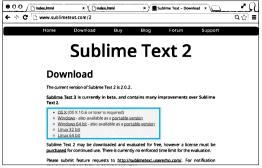



図4-8 ダウンロードページ

ダウンロードが終ったら、ダウンロードしたファイルを各OSでダブルクリックしてインストーラを立ち上げてください。環境によって多少の違いがあるかもしれませんが、基本的にはインストールウィザードをクリックして進めていくだけです。起動すると次のような画面が現れます。

図4-9 Sublime Text

インターネットでは、Sublime Textのメニューを日本語化する方法もいくつか紹介されています。バージョンによって方法は異なりますが、メニューを日本語化するプラグインが提供されています。「sublime text 日本語」などのキーワードで検索してみてもよいでしょう。

これでプログラミングの準備が整いました。さっそく書きはじめたいところですが、まず次章ではJavaScriptの基本的な文法をおさえておくことにします。

Chapter 05

JavaScriptにちょっと入門

本章で勉強すること

- ゲーム作りに欠かせないプログラミング言語について勉強しよう
- JavaScriptはWebアプリケーションでは最も普及した言語です
- ブラウザでJavaScriptを実行する方法を学ぼう
- 変数、関数、配列などプログラムの基本となる要素を解説します
- プログラムにおけるオブジェクトについても簡単に理解しておきましょう
- 制御構文を利用してプログラムの流れを作ろう

JavaScriptとは

本書で使うJavaScriptはWebページの挙動を制御するために開発されたプログラミング言語です。そのため、JavaScriptはWebページ内に書いて動作させることができ、HTMLやCSSをプログラムによって操作できるようになっています。

JavaScriptは1995年、Netscape Navigatorというブラウザのためにブレンダン・アイク氏によってほとんど1ヶ月くらいで開発されました。その当時は、ここまで普及するとは思っていなかったでしょうが、現在はWeb技術の世界では最も一般的な言語となっており、さまざまな使われ方をしています。

Netscape社のブラウザ開発はすでに終了しており、製品としては存在していません。しかし、Firefoxブラウザは Netscape Navigatorを引き継いだフリーのソフトウェアであり、Netscape Navigatorの孫のような存在ともいえるでしょう。

JavaとJavaScriptの関係性

05

コンピュータ技術の世界にはJavaという普及した言語があり、JavaScriptと何か関係がありそうに思えますが、まったく別物です。基本的にJavaScriptとJava は「風車」と「牛」と同じ関係性を持っています。つまり名前が多少かぶっているというだけで、ほかにあまり関係性はありません。

Java <> JavaScript
うし <> ふうしゃ

図5-1 JavaとJavaScript

両者はプログラミングに使われる言語ですが、プログラム言語の世界では名前と技術的な中身が一致していないこともよくあります。たとえば、JavaScriptよりC#のほうがJavaに近い言語です。しかし、C#はC言語やC++とはだいぶ異なっています。プログラミング言語の世界では、名前からその言語の性質を単純に判断しないように心がけましょう。

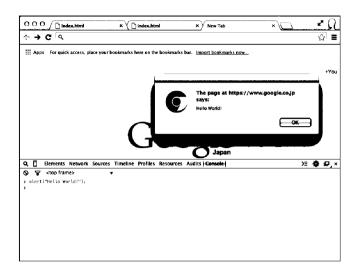
初めてのJavaScript「Hello World」

プログラマーの世界で最初に何か新しい言語やフレームワークを使うときに、一番最初にやってみるのは「Hello World」という文字を出力することです。べつにほかのことを試してみてもよいのですが、ある種の伝統にもなっていますので、今回もそれを守って「Hello World」を出してみましょう。

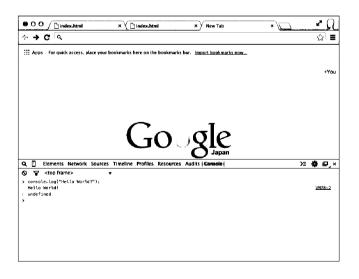
JavaScriptはChromeブラウザのデバッグツールから試すことができます。Macintoshの場合は[Command] ＋[Option]＋[i] でChromeのデバッグツールを開けます。Windowsの場合は[f12]または[Ctrl]＋[Shift]＋[i]を押してください。デバッグツールが立ち上がったら、一番右のコンソールのタブを開いてください。

最初の「Hello World」はブラウザのポップアップで出してみることにしましょう。コンソールのコマンドラインに次の一文を書いて［enter］キーを押してください。

```
alert(“Hello World!” );
```

［enter］キーを押すと、次のようなポップアップが出るはずです。

次は「Hello World」をコンソール自体に出力してみたいと思います。実際の開発ではテストや確認の目的で、データをコンソールに出力することが多くあります。


```
console.log(“Hello World!” );
```

<script>タグ

コマンドラインからJavaScriptのコードを入力した場合、実行されるたびにコードが消えてしまうのでとても不便です。また、コードを配布することもできませんので、今度はファイルにJavaScriptを保存してみましょう。一番簡単なやり方はHTMLの中にJavaScriptを書くことです。そのためにはHTMLの<script>タグを使います。

```
<html>
    <head>
        <script>
            alert(“Hello World”);
        </script>
    </head>
略
```


このコードを例えばhello.htmlという名前（後ろが.htmlとなっていれば名前は何でも大丈夫です）で保存して、ブラウザにドラッグ＆ドロップすると次のようになります。

このようにJavaScriptはHTMLの中では<script>タグから</script>のあいだに書くのが基本です。でも、ゲームのようにプログラムが長くなる場合や作ったプログラムをほかの人と共有したいときもあるかもしれません。その場合は、HTMLに埋め込むのではなく、別のファイルに書いたJavaScriptをHTMLの中に読み込むようにします。たとえば、次のような内容のmyScript.jsというファイルがあったとします。

```
alert(“hello world”);
```

この場合は、HTMLファイルの<head>タグから</head>タグのあいだで次のように読み込みます。

```
<html>
<head>
    <script src="myScript.js"></script>
</head>
略
```

作ったHTMLファイルとmyScript.jsとを同じところに置き、実行してみるとさきほどと同じ結果が得られます。この本で扱うenchant.jsも1つの大きなファイルにまとまっていますので、enchant.jsを使う場合には上の方法でファイルを読み込む必要があります。

本書はJavaScriptの入門書ではないので、あまり詳しい文法を説明することはできませんが、以降ではゲームの開発に関係ありそうなJavaScriptの機能や文法を説明していきます。

console.log

先のHello Worldでも使っていたのですが、ページ上には現れない出力をデバッガのコンソールに出すことができます。プログラムがちゃんと動いているか、自分の思ったとおりのデータになっているかはこの方法で調べたほうが便利です。これにはconsole.logという命令文を使用します。

```
console.log( "Hello World" );

console.log(7);
console.log( [ 1, 2, 3, 4, 5, ] );
console.log( { hp: 100, mp: 50 } );
```

文字

数字

配列（配列についてはのちほど説明します）

オブジェクト（オブジェクトについてものちほど説明します）

```
Elements  Network  Sources  Timeline  Profiles  Resources  Audits  Console
<top frame> ▼  Preserve log
> console.log("Hello World");
  console.log(7);
  console.log( [ 1, 2, 3, 4, 5, ] );
  console.log( { hp: 100, mp: 50 } );
  Hello World
  7
  [1, 2, 3, 4, 5]
  Object {hp: 100, mp: 50}
<· undefined
>
```

`console.log`は、カンマで複数を同時に出力することも可能です。

```
console.log(“Hello World” , 7, [ 1, 2, 3, 4, 5, ], { hp: 100, mp: 50 });
```

変数 (var, variable)

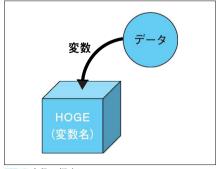
変数は名前の付いている箱のようなものです、中に計算結果やプレイヤーからの入力など、値が変化するものを入れて扱うことができます。

```
var hoge = “Hello World” ;
console.log( hoge );
```

右の単語は変数名です。変数に右のデータ（Hello World）を入れています

コンソールに変数hogeの中味を表示させます。`console.log( “Hello World” );`と同じ意味になります

図5-2 変数の概念

05

ここでは「=」(イコール)を使っていますがこれは、数学における「等しい」という意味とは異なります。JavaScriptの=は(多くのほかのプログラミング言語もそうですが)、「変数等に値を入れる(代入する)」という意味になります。

たとえば変数hogeに3を加える場合は、hoge = hoge + 3と書きます。足し算の結果、変数hogeの値は3増えることになります。こういう変数等の値を操作する記号を**代入演算子**と呼びます。

ちなみにこの記述は、「+=」という別の代入演算子を使ってhoge += 3と書くこともできます。変数に足し算をする処理はよくあるので、このほうが簡潔です。

▼ 表5-1 代入演算子(一部)

記号	意味	使用例
+=	足して代入	a += b (a = a + b と同じ)
-=	引いて代入	a -= b (a = a - b と同じ)
*=	掛けて代入	a *= b (a = a * b と同じ)
/=	割って代入	a /= b (a = a / b と同じ)
%=	割った余りを代入	a %= b (a = a % b と同じ)

また、数字を1だけ増やしたいときは、hoge++や++hogeという書き方も可能です。どちらもhogeの値を1つ増やします。逆に1つ減らしたい場合は、hoge--や--hogeと書いてください。どちらの書き方も単独で使うときは同じ意味になりますが、書き方によっては動作が異なる場合もあります。

たとえば、hoge = 5のとき、fuga = hoge++とすると変数hogeの値は6でfugaの値は5になります。この場合はhogeの値をfugaに代入してからhogeの値を1つ増やしています。一方、fuga = ++hogeと描いた場合は先にhogeの値を増やします。ですので、この場合はhogeもfugaも6になります。微妙な違いですが簡単に頭に入れておいてください。

関数(function)

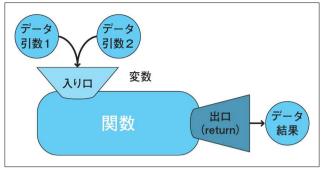
長いプログラムを書いていると同じような処理を何回も繰り返すことがあります。そういうときは処理を**関数**にしておくと簡単に呼び出すことができます。今まで使っていたconsole.logやalertもJavaScriptの関数です。

関数には入力と出力があります。これは引数と戻り値とも呼ばれます。関数に引数を与えると、関数の中で計算や処理を行って、戻り値として返してくれます。中でどんな処理をしているかは具体的に知る必要はないので、プログラムをすっきりとまとめることができます。

図5-3 関数の概念

```
function baiNiSuru( nyuryoku ) {
    var syuturyoku = nyuryoku * 2;
    return syuturyoku;
}
```

引数を2倍にして返す「倍返し」の関数を作ってみましょう

returnは関数の出力を示しています。この値が戻り値として返ります

作った関数は次のように使えます。

```
var hoge = baiNiSuru( 5 );
console.log( hoge ); // 10

var hogeNoBai = baiNiSuru( hoge );
console.log( hoge, hogeNoBai ); // 10, 20
```

5という値を2倍にして、変数hogeに入れます

さらにhogeを2倍にして、変数hogeNoBaiに入れます

JavaScriptを記述するときのルール

命名規則

変数と関数の名前はアルファベットあるいは$または_（アンダースコア）で始めます、名前の中に数字を含めても構いませんが、一番最初の文字に数字を使ってはいけません。特別な文字（!”#%&’?><など）は変数や関数の名前には使えません。また、大文字と小文字は区別されるのでhogeとHogeは別の変数です。

```
hoge    →OK
Hoge    →OK（習慣的に、頭文字を大文字にするとクラスを示します）
hoGE    →OK
hoge2   →OK
_hoge   →OK
$hoge   →OK
hoge!   →NG
2hoge   →NG
ho#ge   →NG
```

;（セミコロン）

今までも登場していましたが、処理を終えたときには最後に「;」（セミコロン）を入れます。日本語で言うと「。」に近いような役割があります。

セミコロンを文の終了に使う言語は多くあります（たとえばC言語、C++、C#、Javaなどです）。しかし、じつはJavaScriptはセミコロンがなくても動作します。JavaScriptではセミコロンは必須の文法ではありませんが、プログラマーの習慣としてセミコロンを使っています。逆に文の区切りにセミコロンを使わない言語から来たプログラマーは「;」を嫌がるので、JavaScriptでも使いたがりません。サーバーで使われるJavaScript(Node.jsなど）では「;」を余計に使うことに反対するプログラマーもいます。これからほかの人が書いたJavaScriptを見る機会があると、「;」を使っている人と使っていない人がいると思いますので、注意してみてください。

コメント

プログラムは基本的にコンピュータに対して命令を出すためのものですが、プログラムを改善するために人間もソースコードを読むことがあります。分かりにくいコードも出てきますので、次にソースコードを読む人のために、**コメント**を残しておくように心がけましょう（ちなみに、その「次の人」は極めて高い確率で数ヶ月先の自分です）。コードだけではすぐに分からないことや忘れそうなものはコメントとして残します。コンピュータはコメントを完全に無視するので、好きなことを書いてもプログラムに影響はありません。

■ショートコメント

1行以内に収めるコメントはショートコメントと呼ばれています。JavaScriptは「//」をマーカーとして使っています。 //の右に書いてあることは実行時にはすべて無視されます。

```
//これは実行しない。
var hoge = 5; //ここからこの行のことは実行しない。
console.log( hoge ); // メモ：hoge の出力は5です。
```

05

■ロングコメント

「/*」で開始し、「*/」で終わります。マーカーに挟まれている部分は改行されていても無視されます。

```
/* コメントです。
まだコメントです。
このあと終ります。*/
```

このように複数行を一気にコメントできますが、ちょっと分かりにくいのでほとんどのプログラマーはロングコメントを次のように書いています。

```
/**
* コメントです。
* まだコメントです。
* このあと終ります。
*/
```

無駄に「*」（アスタリスク）を書いていますが、マシンから見ると途中の「*」はすべてロングコメントの中に入っているので、実際には意味を持ちません。しかし、コメントは人間のためのものですから、コメントを分かりやすくするためにマーカーを入れています。

■コメントアウト

一時的に何かの実行を止めたいときはコメントアウトをすることがあります。たとえば次のようなコードがあったとします。

```
var hoge = 5;
console.log( hoge );
```

console.logを実行したくないときは、これをコメントアウトしましょう。

```
var hoge = 5;
//console.log( hoge );
```


デバッグやテストのときには、こうしたコメントアウトをよく行います。

コードブロック

さきほど関数を書いているときに出てきていますが、JavaScriptではコードのまとまりを{ ～ }で囲みます。関数を書くときにはこのコードブロックを使います。

```
function doHoge( x ) {
    var y = (x + 4) *2;
    return y;
};
```

のちほど説明する制御構文を書くときにも基本的にはコードブロックを使います。ブロックの中身が1行しかないときは{ ～ }はなくても大丈夫です。このため、１行のときはコードブロックを書かないとい

う人もいますが、実際にはとても危険です。あとから別の人が不用意に1行を足したとすると、プログラムがうまく動作しなくなる可能性があります。このトラブルはよくありますので、注意してください。

JavaScriptの型

変数には**型**というものがあります。これはデータの種類を表すもので、たとえば数字の計算をしたければ数値型の変数、文字を扱いたければ文字列型の変数を用います。たとえば、数値型の1+1は２ですが、文字列型の"1"+"1"は"11"（文字列は"や'で囲みます）と文字と文字をくっつけたものになり、扱いが変化します。

文字列（String）

文字列型はクォテーション（"や'）で囲まれた文字列を与えることで作れます。

```
var charaName = "キャプテン";
var charaName = 'キャプテン';
```

ダブルクォテーション（"）
シングルクォテーション（'）

05

文字列中でシングルクォテーションやダブルクォテーションを使いたい場合は次のように書いてください。

```
var answer = "It's ok";

var answer = "He is called '太郎'";
var answer = 'He is called "太郎"';
```

ダブルクォテーション中でシングルクォテーションを使う
ダブルクォテーション中でシングルクォテーションを使う
シングルクォテーション中でダブルクォテーションを使う

数字（Number, Integer, Float, Double）

JavaScriptでは数字を変数に与えることで数値型の変数が作れます。

```
var x1 = 34.00;
var x2 = 34;
```

小数点付き
小数点なし

ブーリアンとブール代数（Boolean）

ブーリアンは論理型ともいわれ、「真 = true」か「偽 = false」という2値しかとらないデータ型です。

```
var x = true;
var y = false;
```

05

ブーリアン型のデータは「==」と「!=」そして「===」と「!==」といった演算子で演算が可能です。たとえば、次のようなコードを見てみましょう。

```
var x = 1 == 1;
console.log( x );
```

値は真(true)

==はその左右の値を比較して一致していれば真（true）を返す演算子です。上の例では1という同じ値を比較していますので、変数xには真（true）が入ります。!=は逆で一致していないときに真が返ります。

```
console.log( 1 == 1 );
console.log( 1 == 2 );
```

値は真(true)
値は偽(false)

```
console.log( 1 != 1 );
console.log( 1 != 2 );
```

値は偽(false)
値は真(true)

ところが、JavaScriptは親切すぎるところがあり、ちょっとした落とし穴があります。次のような例を見てください。

```
console.log( 1 == "1" );
```

値は真(true)

この場合は数値型の1と文字列型の"1"を比較しているので、厳密にいえば真にはならないはずなのですが、JavaScriptは親切にも真を返してきます。型も同じかどうかを知りたいときは===と!==の演算子を使います（どちらも通常より1つ=が多い）。こうすればデータの型まで一致するか確認するようになります。

05

```
console.log( 1 === "1" );

console.log( 1 === "2" );

console.log( 1 !== "1" );
console.log( 1 !== "2" );
```

値は偽(false)
値は偽(false)
値は真(true)
値は真(true)

Undefined（未定義）とNull（ヌル）

両方とも何も変数がないことを示していますが、意味上の違いがあります。

undefinedは「未定義」の意味で、null（「ヌル」と呼びます）は「定義したが、値はない」ことを意味しています。初めてundefinedとnullを見た方はすこし戸惑うかもしれませんが、ゲームを作るうえではあまり影響はありません。

```
var cars;
person = null;
```

変数は未定義

変数personには値がない

配列 (Array)

配列は変数に近いものですが、１つの箱ではなく、複数の箱がひとまとまりになったものです。

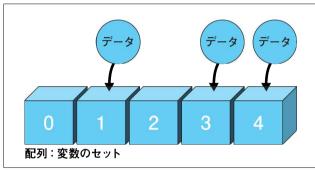



図5-4 配列の概念

このようにひとつの変数名で複数のデータを管理できます。

```
var hairetu = [1,2,3,4];
```

配列のデータの型は全部同じ型である必要はありませんが、通常は同じ型のものをまとめます。

```
var hairetu = [1, "2", "三"];
```

配列のデータにアクセスするにはインデックス（index）という番号でどの箱かを指定します。JavaScript、そしてほとんどの言語の配列のインデックスは0から始まります。日常的には変ですが、コンピュータの世界では0で始まるケースがよくあります。

```
var message = [ "Hello" , "Array" ];

console.log( message[0] );
console.log( message[1] );
```

文字列"Hello"が入っている

文字列"array"が入っている

オブジェクト (Object)

オブジェクトはデータや関数をひとつにまとめるための型です。JavaScriptの世界では配列もオブジェクトの一種です。ちょっと例を見てみましょう。

```
var hero = {
    name: "キャプテン",
    hp: 100,
    mp: 50,
};
```

オブジェクトに紐付けられたデータは変数とは呼びません（実際は変数と同じですが）、オブジェクトに入っているデータは**プロパティ**（property）と呼ばれています。先の例ではheroオブジェクトの中に、name、hp、mpというプロパティが用意されています。

プロパティにアクセスするにはオブジェクト名の後ろに、「.」（ドット）を付けて表現します。

```
console.log("hero HP:", hero.hp);
```

heroオブジェクトのhpプロパティの値が表示される

さらにオブジェクトには関数も紐付けられます。この場合も名前は関数ではなく**メソッド**（method）と呼ばれます。

今まで使ってきたconsole.log()の.log()の部分はconsoleのオブジェクトに紐付けられたlogメソッドでした。

自分でオブジェクトにメソッドを追加したいときはfunctionを使って定義してください。getKougek

iRyokuという関数をメソッドとして定義したい場合は次のようにします。

```
var hero = {
    function getKougekiRyoku( buki ) {
        var kougekiRyokuGoukei = this.attack + buki.attack;
        return kougekiRyokuGoukei;
}
};
```

オブジェクトのパラメータとメソッドは「オブジェクト」と「パラメータまたはメソッド名」を「.」で区切って表現しますが、もうひとつ方法があります。それは配列のようにオブジェクト["パラメータまたはメソッド名"]と書く方法です。

```
console.log("hero HP:", hero["hp"]);
```


この書き方はあまり分かりやすいコードではありませんが、「パラメータまたはメソッド名」を変数に入れて、その変数でオブジェクトにアクセスしたいときに便利です。

```
var parameter = "hp";                          // hero hp 100と出力される
console.log("hero", parameter, hero[parameter]);

var parameter = "mp";                          // hero mp 50と出力される
console.log("hero", parameter, hero[parameter]);
```

typeof

変数などの型を調べたいときはtypeofを使います。

```
console.log(typeof "太郎")
console.log(typeof 3.14)
console.log(typeof false)
console.log(typeof [1,2,3,4])
console.log(typeof {name:'太郎', age:20})
```

文字列型
数値型
ブーリアン
オブジェクト
オブジェクト

typeofを使うとき結果は文字列で返ってきますが、それを変数にも入れることもできます。

```
var hoge = 5;
var hogeType = typeof hoge;
console.log(hogeType == "string")
console.log(hogeType == "number")
```

偽（false）
真（true）

スコープ（Scope）

スコープには「空間」または「範囲」という意味があります。これは関数の使える範囲を限定するものです。

プログラムでは基本的にブロックや関数の内部の変数に外からアクセスすることはできません。逆にブロックや関数の内部から外の変数は使うことができます。ちょっと例を見てみましょう。

```
var fruitName = " オレンジ";
function myFunction() {
}
```

この中は**fruitName**変数が使えます
この中は**fruitName**変数が使えます

この場合、関数の中から外側の変数**fruitName**は利用できます。しかし関数の中で定義した変数に外からアクセスすることはできません。

```
function myFunction() {
    var fruitName = "オレンジ";
    この中はfruitName変数が使えます
}
外ではfruitName変数が使えません
```

プログラムの一番広いスコープは**グローバルスコープ**と呼ばれています。そしてグローバル以外は**ローカルスコープ**と呼ばれます。プログラムを書くときは極力ローカルスコープを使うようにしましょう。理由は複数ありますが、おもなものは次のとおりです。

1. **同じスコープで名前が重複するとバグが起きやすい**
2. **パフォーマンスの面からいえば、ひとつ外のスコープの変数を探すには時間がかかる**
3. **ほかのフレームワーク、ライブラリ、ツールなどと変数名が重複することを防げる**

グローバル変数を使うときは、その影響についてよく考えてからにしましょう。

JavaScriptの制御構文

今まで見てきたJavaScriptで計算などはできると思いますが、まだ条件に応じてプログラムの流れを作ることはできません。プログラムの流れを作るには制御構文を使う必要があります。JavaScriptで使える制御構文は次のとおりです。

break	for ～ in	throw
catch	function	try
continue	if ～ else	var
do ～ while	return	while
for	switch ～ case	

すべてを理解する必要はありませんが、最低限if ～ else、for、for ～ in、switch ～ case、break、continue、whileを覚えておけばよいでしょう。

if ～ else

ある条件を調べて、真（true）だったら実行し、偽（false）だったらelse以下を実行します。

```
if ( 1 + 1 == 2 ) {        // この条件では常に真(true)になります
    console.log( "OK" );
} else {                   // こちらの行が実行されることはありません
    console.log( "NG" );
}
```

for

処理を限られた回数繰り返したいときに使います。

```
for(var i=0; i < 10; i++) {   // 変数iが0から9までのあいだ、10回繰り返します
    console.log( "count" , i);
}
```

05

for ~ in

とある配列またはオブジェクトに対してループを行います

```
var data =  [ "a" , "b" , "c" ];
for (var i in data) {
    console.log( "index" , i, ":" , data[i] );
}
```

index 0 : a
index 1 : b
index 2 : c

switch ~ case、defaultとbreak

複数の条件を判定します。caseはswitchの条件の分岐、そしてどの条件にも当てはまらないデータはdefaultの分岐に入ります。breakが実行されると、条件分岐の処理を終了して、次に進みます。

```
switch(x) {
    case 0:
        console.log( "zero" );
        break;
    case 1:
        console.log( "one" );
        break;
    default:
        console.log( "other" );
        break;
}
```

xが0だった場合の処理
xが1だった場合の処理
それ以外の場合の処理

breakとcontinue

breakでループを中断します。

```
for(var i=0; i < 5; i++) {
    if (i == 3) {
        break;
    }
    console.log( "count" , i);
}
```

構文上はiが0から4になるまでループするようになっているが
iが3の時点でループを抜ける
iが1、2までは実行される

continueはループのスキップをします。

```
for(var i=0; i < 5; i++) {
    if (i == 3) {
        continue;
    }
    console.log( "count" , i);
}
```

iが3のときはループをスキップする
1、2、4、5が出力される

while

forのようにループをします。forでもwhileも同じことはできますが、forを見ることが多いようです。

```
var i=0
while(i < 10) {
     i++;
     console.log(i)
}
```

0、1、2、3、4、5、6、7、8、9が出力される

JavaScriptをもっと知りたいあなたへ!

この章では本書で紹介したコードを読むのに便利なようにJavaScriptについていろいろと説明しましたが、急ぎ足でしたので初めてプログラミング言語に触れる人には、理解しきれないことも残ってしまうかもしれません。しかし、JavaScriptを勉強してみると、さらにいろいろなことが見えてくるはずです。ここではまだJavaScriptの一部しか説明できていませんが、JavaScriptをマスターするための本はたくさんありますので参考にしてください。

プログラミング言語の世界には
いろいろ奥深いものがあります

「JavaScriptの絵本」
(株)アンク 著、翔泳社、ISBN978-4-7981-2617-3

05

Part

2 ゲームの基本機能を作る

Chapter 06

enchant.jsの環境を整備

enchant.jsをゲットして、
アーカイブを展開します

フレームワークを読み込んで、
サンプルプログラムの動作を確
認します

ブラウザのデバッグツールで
エラーがないことを確認したら、
いよいよプログラミング開始です

本章で勉強すること

- フレームワークはゲームの開発を助けてくれる強い味方
- 作るプログラムやゲームの種類によっていろいろなフレームワークがある
- 自分の作りたいものに合ったフレームワークを選ぼう
- enchant.jsフレームワークをインストールして試してみよう

フレームワークとツールって？

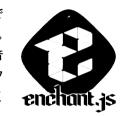
フレームワーク

フレームワークというのはどんなものでしょうか？ オンライン百科事典のWikipediaによれば、フレ

ームワークは次のように説明されていました。

フレームワーク（framework）

開発・運用・意思決定を行う際に、その基礎となる規則・構造・アイデア・思想などの集合のこと。日本語では「枠組み」などと訳されることが多い。—wikipedia（http://ja.wikipedia.org/wiki/フレームワーク）

コンピュータの世界におけるフレームワークは、あるタイプのソフトウェアでよく使う機能をあらかじめプログラムして、使いやすく提供してくれるものです。たとえば、ゲームのフレームワークであれば、ゲームでよく使う画面の制御や音声の機能を簡単に利用できるようにしてくれます。プログラムの「枠組み」（フレームワーク）だけが用意されているので、その枠組みに従うことで簡単に書くことができ、プログラムの見通しを良くし、エラーの予防にもなります。

また、フレームワークの助けを借りることで、1つのソースコードからPC用とスマートフォン用のプログラムを作り分けることもできます。もちろん、すべてのケースでそのような作り分けが簡単にできるわけではありませんが、開発環境の違いを吸収することもフレームワークの役割のひとつです。

ツール

一方、**ツール**はプログラミングやソフト作りを助けてくれる各種のソフトウェアのことです。**プログラミングツール**ともいいます。

プログラミングツール（programming tool）

プログラミングツールまたはソフトウェア開発ツールとは、ソフトウェア開発者がプログラムやアプリケーションを作成・デバッグ・保守・サポートするためのソフトウェアである。一般に比較的単純なプログラムを指し、それらを複数組み合わせて作業を行う。

—wikipedia（http://ja.wikipedia.org/wiki/プログラミングツール）

たとえば、ソフトウェアのエラーを調べて原因を特定する手がかりを与えてくれるデバッガや、ソフトウェアの性能を調べるツール、テストツール、ソースコードの変更履歴を管理してくれるツールなどもあります。これらを組み合わせた統合開発環境（IDE：Integrated Development Environment）もありますが、この本ではそこまで大がかりなツールは使いません。

enchant.js

今回のゲームを作るためにはどんなフレームワークがよいのか、いろいろ検討しました。

2～3年前と比べてもフレームワークはどんどん増えているので、どのフレームワークを選ぶかだけでも、深く悩みそうです。そんな場合はとにかく話を前に進めるため、「自分が今作りたいゲーム」と「自分が今開発に使える時間」をよく考えたうえで、その条件に当てはまるフレームワークをざっくりと絞ってしまいましょう。

もちろん、そうしてみても複数の候補は残ります。そして、いろいろ調べたとしても自分のやりたいことと完璧に一致するフレームワークは基本的にないと思ってください。どのフレームワークにもたくさんの強みと弱みがありますので、ずっと悩み続けるよりはどれかを選んで前に進むことが肝心です。フレームワークを使ってみればそのフレームワークの特質がいろいろと見えてくるので、自分の要件に合わなければそのときにフレームワークの変更も考えましょう。

Webブラウザなどで「フレームワーク」「ゲーム」などのキーワードで検索してみると、さまざまな情報が得られます。また、ゲームに特化したフレームワークは「ゲームエンジン」というキーワードでも紹介されています。
ゲーム用フレームワークは開発するゲームの種類やプログラムが動作するプラットフォームによって分類が可能です。たとえば、3DゲームをつくるためにはUnityやUnreal Engineなどが知られています。また、ブラウザゲームであれば本書でも取り扱うenchant.js、gameQueryなどが有力でしょうか？ノベルゲームに特化したフリーの吉里吉里やNScripterもあります。
これらのゲームエンジンでスマートフォンのゲームを作ることもできますが、スマートフォン向けのゲームエンジンとしてはCocos2d-xやSprite Kitなどが知られています。

今回のプロジェクトに必要なのは、無料で2Dゲームが作れ、できたゲームを簡単にシェアできることです。そこで今回はスマートフォン専用のフレームワークを外すことにしました。スマートフォンアプリケーションの開発は（基本的にちょっとですが）お金がかかるものが多いので、ブラウザ系のフレームワークに絞りました。ブラウザ用のフレームワークなら、WindowsやMacintoshなどのPCやスマートフォンでも同じように動作する2Dゲームが作れます。

こうやって絞り込んでも、有力なブラウザ系フレームワークだけでも候補は10点近くにのぼりました。それぞれ、「パフォーマンス」、「高度な開発にも対応している」、「3Dもできる」、「使いやすさ」などにフォーカスしていて特徴があります。今回はおもに使いやすさと／分かりやすさでenchant.jsを選びました。

enchant.jsのWebサイト
http://enchantjs.com/ja/

enchant.jsのWebサイトでは、その特徴を次のように紹介しています。

1. カンタンにゲームやアプリを開発できるHTML5 + JavaScriptフレームワークです。
2. 2011年に公開され、すでに 1,000 本以上のゲーム/アプリが公開されています。
3. オープンソース（MITライセンス）で、無料で利用できます。
4. ドキュメント・書籍・チュートリアルサイトが充実しています。
5. たくさんのプラグインで機能を拡張できます。
6. UEI/ARC を中心としたメンバによって開発・メンテナンスされています。
7. プログラミング教育のためにも利用されています。

この中から、本書としては7を重視し、2の実績も考慮してenchant.jsを選択しました。

実際にほかのフレームワークも試してみましたが、enchant.jsはシンプルでも2Dゲームに必要とされる機能はしっかりと備わっています。普通の2Dのゲームであれば短時間で開発ができるのは確かです。

enchant.js をインストール

enchant.jsはまだ開発が継続されているフレームワークのため、頻繁にバージョンアップされています。最新バージョンは、いつでもWebサイトからダウンロードできますが、今回作るゲームについては統一したバージョンで開発を進めるものとします。

本書のダウンロードサイトからPirateTactics.zipをダウンロードしましょう。このアーカイブの中に今回利用するenchant.jsのバージョン0.8.2とゲームのソースコードなどが入っています。

本書サンプルダウンロードページ
http://www.shoeisha.co.jp/book/download/9784798137841

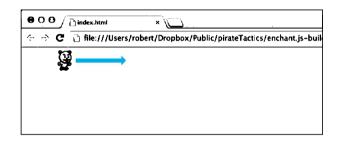
PirateTactics.zipがダウンロードできたら、アーカイブを解凍して、フォルダをまるごとコピーしましょう。一時的ならデスクトップに展開しても構いません。

enchant.js の動作チェック

新しいフレームワークやツールを使うときは、基本の動作チェックが欠かせません。そのため、まずはコピーしたenchant.jsに付いているサンプルを起動してみましょう。

デスクトップに解凍した場合は、enchant.js-builds-0.8.2-b → examples → beginner → hellobear とフォルダを開くとindex.htmlがあります。これをWebブラウザ（今回はGoogle Chrome）にドラッグ＆ドロップします。すると、ブラウザの画面に、歩いているクマが現れます。

このようにクマが見えて無事に歩けば、enchant.jsの場合、とりあえず動作に必要な環境はクリアしていると判断してよいでしょう。

06

デバッグツールを使おう

クマがちゃんと歩いたら基本的には問題ないはずですが、もっと動作を確実に確かめるためにブラウザのデバッグ機能を使ってみることにします。Chromeのデバッグ機能はChromeに標準で備わっているプログラミングツールのひとつで、エラーが有るか無いか、どこにエラーが起きたかを簡単に確認できます。

デバッグツールを立ち上げるにはChromeの開発環境を開く必要があります。Mac OSの場合［Command］＋［Option］＋［i］のキーを同時押しすると開きます。Windowsの場合は［F12］キーまたは［Ctrl］＋［Shift］＋［i］です。

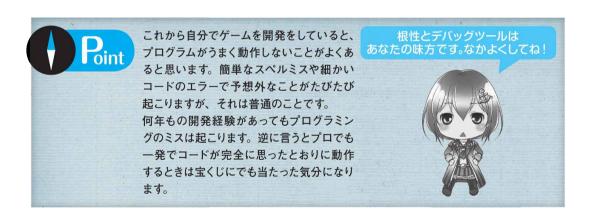
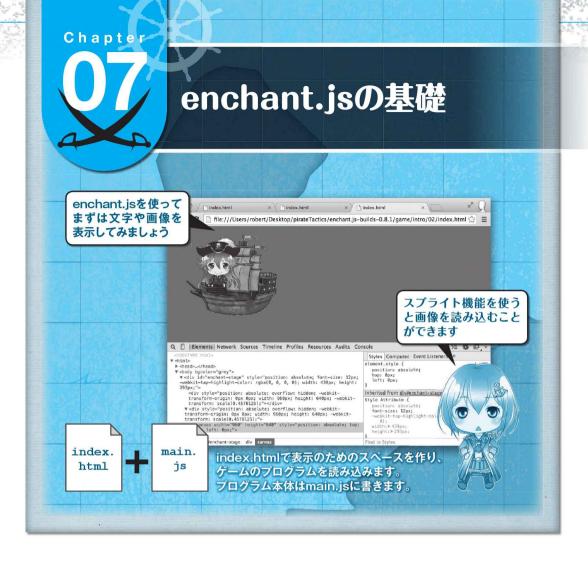
バグという言葉の語源は文字どおり英語のBug（虫）から来ています。しかし、この言葉はコンピュータの登場以前から、機械の原因不明な不具合を表す言葉として技術者のあいだで使われていました。

バグ→プログラムの不具合
デバッグ→バグを解消して不具合を治すこと
エンバグ→間違えてバグをプログラムに加えてしまうこと

ここで一番右にあるデバッグツールのコンソールのタブを開いてみて、とくに赤いエラーメッセージが出なければ問題ありません。

同じexamplesフォルダには、ほかにもさまざまなサンプルがありますので、確認のためにひととおり動作させてみるとよいかもしれません。

これから自分でゲームを開発をしていると、プログラムがうまく動作しないことがよくあると思います。簡単なスペルミスや細かいコードのエラーで予想外なことがたびたび起こりますが、それは普通のことです。
何年もの開発経験があってもプログラミングのミスは起こります。逆に言うとプロでも一発でコードが完全に思ったとおりに動作するときは宝くじにでも当たった気分になります。

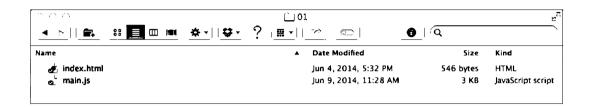
07

本章で勉強すること

- 最低限のenchant.jsのサンプルで、基本を勉強していこう
- index.htmlとmain.jsを用意
- enchant.jsの読み込み方と動作の確認方法を学ぼう
- 文字の表示はラベルを使う。ラベルを貼ってみよう
- いよいよキャラクターが画面に登場
- 外部のファイルを読み込んで、スプライトを使ってみよう

では最初にenchant.jsの実行できる最低限のサンプルを作りながら基礎を身につけていきましょう。

まず本章用のサンプルコードのフォルダを開きます。アーカイブを展開したフォルダから、enchant.js-builds-0.8.2-b → game → intro → 00と辿ってフォルダを開いてください。

今回のようにenchant.jsで小規模なゲームを作る場合、プロジェクト（enchant.jsだとフォルダとファイルだけで構成されます）は基本的にとてもシンプルです。画像ファイルやサウンドを除けばほとんど2つのファイルで構成されています。

サンプルフォルダにはその2つのファイルがあります。index.htmlとmain.jsです。それぞれについて見てみることにしましょう。

index.html

ひとつ目のindex.htmlはものすごくシンプルなHTMLで書かれたWebページです。enchant.jsで作られたゲームを表示する領域を定義しているだけです。

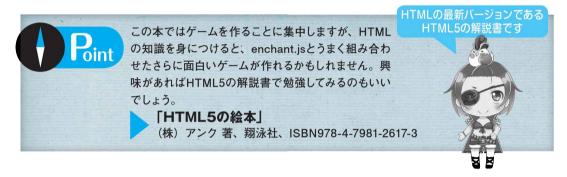
今回のゲームくらいであれば、このようなシンプルなHTMLで十分です。また、このような単純なHTMLのタグでenchant.jsがブラウザ中に表示できるということは、ほかのWebサイトのHTMLページの一部としてゲームを組み込むのも容易だということです。たとえば、自分のホームページがあれば、そのページの中で作ったゲームを動かすこともできます。

ではindex.htmlの中身を確認してみましょう。

▼ index.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta http-equiv="x-ua-compatible" content="IE=Edge">
    <meta name="viewport" content="width=device-width, user-scalable=no">
    <script type="text/javascript" src="../../../../build/enchant.js"></script>
    <script type="text/javascript" src="main.js"></script>
    <style type="text/css">
        body {
            margin: 0;
            padding: 0;
```

```
            }
        </style>
    </head>
    <body bgcolor="grey">
    </body>
    </html>
```

非常に短いですが、この中にはいろいろな要素が含まれています。

このままだとちょっと分かりにくいと思いますので、いったん中身を無視して骨組みだけを見てみましょう。index.htmlを分解して主要なHTMLのタグだけにすると、次のようになります。

```
<!DOCTYPE html>
<html>
<head>
中略
</head>
<body bgcolor="grey">
</body>
</html>
```

HTML5の形式に則って基本的なタグだけが書かれています。これはenchant.jsのアプリケーションというよりは、HTML5のWebページの基本的な形です。まず先頭行でHTML5のページであることが宣言され、<head> ～ </head>タグのあいだで、enchant.jsに関するいろいろな設定が書かれます。<body bgcolor="grey">タグと</body>タグに挟まれた部分は本来ならページの本文に相当するものが書かれますが、ゲームの画面はenchant.jsが表示しますので何も書かれていません。bgcolor="grey"で背

景の色をグレーに指定します。

ではもういちど、元のソースを解読しながら今度は<head> ~ </head>タグのあいだで書かれていることを確認しましょう。

ここではテキストの文字コードをUTF-8に設定しています

Internet Explorerとの互換性を保つための指定です

これで画面の比率を固定します

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="x-ua-compatible" content="IE=Edge">
    <meta name="viewport" content="width=device-width, user-scalable=no">
    <script type="text/javascript" src="../../../../build/enchant.js"></script>
    <script type="text/javascript" src="main.js"></script>
    <style type="text/css">
        body {
            margin: 0;
            padding: 0;
        }
    </style>
</head>
```

これはゲームのコードmain.jsの配置です

enchant.jsフレームワークの読み込み

一番大切なポイントは次の2行だけです。それ以外に関しては、ゲームの作成には直接関わらない部分ですので、おまじないだと思って書いておけばかまいません（詳細な意味については時間があるときに調べていただいてもよいでしょう）。

```
<script type="text/javascript" src="../../../../build/enchant.js"></script>
<script type="text/javascript" src="main.js"></script>
```

1行目はenchant.jsの読み込みを指定しています。src="../../../build/../enchant.js"でenchant.js本体の場所を示し、type="text/javascript"でJavaScriptのソースであることを示しています。../の部分は今の（index.htmlのある）フォルダの1つ上を示します。../../../../build/enchant.jsとありますので、「index.htmlのあるフォルダの4つ上のbuild以下にあるenchant.jsというファイルを読み込む」という指示になるわけです（本当にそこにファイルがありますので確認してください）。

これでenchant.jsが利用可能になります。「新しいゲームを作って動かしてみても何も起こらない」というような場合は、デバッグツールを開いてみるか、まずはこちらの指定を確認してみるとよいでしょう。

07

今後別のゲームを作ることになったら、違うフォルダ構成を採用するかもしれません。そのときはこの部分の指定を変更します。

　その次の1行はこれから書くゲーム本体のメインプログラムmain.jsの読み込みを指定します。それでは、main.jsの内容について次に見ていきましょう。

main.js

　main.js を開きましょう。ここにはゲームのコードが書かれていますが、まだ初めの初めですのでごく基本的なことしか書かれていません。

▼ main.js

> enchant.jsを使う前にフレームワークを有効にするために必要な処理

```
enchant();
```

> ページがロードされた際に実行される関数。すべての処理はページがロードされてから行うため、window.onloadの中で実行する。特に new Core(); は、<body> タグが存在しないとエラーになるので注意

```
window.onload = function(){
```

> Core オブジェクトを作成する。画面の大きさは960ピクセル×640ピクセルに設定する

> この大きさだとモバイル端末でも遊べます

```
    var game = new Core(960, 640);
```

> ゲームにシーンを追加

```
    var sceneGameMain = new Scene();
    game.pushScene(sceneGameMain);

    game.start();
};
```

最初はenchant()という初期化のための関数を実行します。enchant.jsを使うときは、この命令をいつも実行させます。実行すると、プログラムを動かすためのenchant.jsのさまざまな準備が整う仕組みになっています。

次はイベントハンドラを書いています。**イベント**とは、プログラムに通知される外部のアクションのことです。ちょっと抽象的な言い方になってしまいましたが、ブラウザゲームであれば、「Webページが読み込まれた」「キーが押された」「ボタンがクリックされた」などの出来事がイベントとしてプログラムに通知されます。これらのイベントが発生したら、その時に行う処理をプログラミングします。「○○が起きたら、××する」というふうにプログラムを書くほうがブラウザのアプリケーションや、ゲームの場合は便利です。こういったプログラミングの仕組みを**イベントループ型**と呼んでいます。ここにはwindow.onloadというイベントハンドラがあります。windowはブラウザのWebページを示しています。ブラウザがページを読み込む（ロードする）と、onloadというイベントが発生します。このイベントが発生してはじめてenchant.jsが安全に使えます。

onloadイベントの前にenchant.jsの機能を使ってしまうと、その部分が実行されるタイミングはブラウザ任せになってしまいます。コードを書いても、動いたり、動かなかったりしますので、かならずonloadイベントのあとに書くようにしましょう。

フレームワークには約束ごとがたくさんあるので、それに従ってプログラムしていってください。

onloadイベントハンドラでは、function()と書いて、その後ろの{ ～ }のあいだにそのイベントで実行させたいことを書いていきます。一番最初にすることはCoreというオブジェクトを作ることです。Coreオブジェクトはゲームプログラムの本体を示すオブジェクトです。

var game = new Core(960, 640)の部分では、new Core(960, 640)でCoreを作り、それにgameという名前を付けています。引数でゲームの画面サイズを決めています。実際の画面はブラウザの表示サイズによって変わりますが、画面の大きさをこのように定義しておけば、サイズ変更はフレームワークが代りにやってくれるのでとても便利です！

画面を960×640にした理由は、その大きさであれば多くのスマートフォンやタブレットで問題なく表示できるからです。

これでゲームの画面ができました。その上にシーン（Scene）を作って、pushScene()という命令でゲームの画面に表示しています。**シーン**はゲームの舞台を意味しています。今回はsceneGameMainというシーンを作りました。この舞台の上にゲームのさまざまな出来事が起こります。ただし、pushSceneを実行しないとシーンはゲームの画面には現れません。複数のシーンを作ってpushScene()で画面を切り替えることも可能です。

フレームワークを読み込み、ページ／ゲーム／シーンができたら`game.start()`を実行します。これで今作っているゲームが実行されます。

メインループはなくなってしまったの？

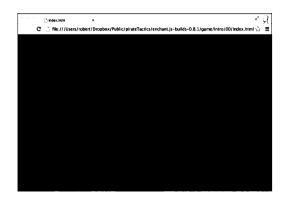
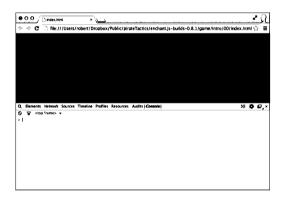
Webブラウザは基本的に非同期のシステムです。非同期とは（大まかにいえば）時間の流れの概念がなく、順序だった動作をしないシステムです。そこで、ブラウザでプログラムを動かすには、説明したようなイベントのやり取りを利用します。「○○というイベントが起ったら、××を実行しよう」という風にプログラムを書くのです。
これはイベントループ（event loop）というプログラムの作り方です。実際、今の時代はブラウザ以外のゲームフレームワークやエンジンもほとんどがイベントループで動作しています。
大昔（といっても実際は数年前ですが）、ゲームを作るときにはメインループ（main loop）というものがありました。メインループのイメージは次のサンプルのような感じです（このコードは実際には動きません、たんなるイメージのコードです）。`while`などの繰り返し命令を使ってゲームが終了するまで、ぐるぐる処理を回すようになっています。

```
function main() {        ← メインループを定義

    initGame()           ← ゲームに必要な初期化処理を行います

    var endGame = false; ← ゲームが終わるまで無限にループします
    while (endGame == false) {

        updateGame();    ← ゲームのロジック処理を更新します


        renderGame();    ← ゲームの描画処理を更新します

        endGame = shouldGameEnd(); ← ゲームの終了条件をチェックをします
    }

    quitGame();          ← ゲームが終了するので終了処理などをしてプログラムを終わります
    exit();
}
```

ただし、近年このようなメインループ式のゲームの書き方は殆ど見なくなりました。筆者がゲーム開発で最後にメインループを書いたのは、たぶん7年くらい前だと思います。その意味では、これからもメインループ式のゲームフレームワークと出会うことはないかもしれません。しばらくはイベントループ式フレームワークの時代が続くと思われます。
今でも、フレームワークやブラウザ、あるいはOSの奥深くでは、上のようなメインループが淡々と仕事をしていると思います。でも、かなり低レベルなプログラミングをしないかぎり、現代ではメインループに出会うことはないでしょう。

07

デバッグツールで確認してみよう

ここまで作ってきたのは、これからゲームを作るうえでの基本となる構成です。まだなにも表示されていませんが、裏ではフレームワークやゲームがちゃんと稼動しています。

今のところ、ゲームの画面には何もありません。見た目には寂しいので、まずデバッグツールを開いて確認をしましょう。Macintoshの場合は［Command］＋［Option］＋［I］キー、Windowsのときは［F12］キーです。

コンソールには見事に何も出ていないのでエラーもないみたいです。でも、エラーがないからといってゲームがちゃんと動いているとは限りません。ほかに確認の手段はないのでしょうか？

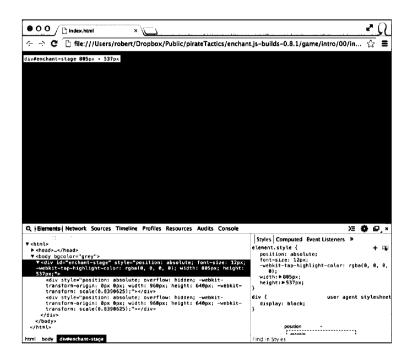
実際にゲームが動いている様子はデバッグツールの「Elements」のタブで確認できます。

「Elements」をよく見ると、ただのindex.htmlにはなかった部分が増えています。

```
<div id="enchant-stage" style="position: absolute; font-size: 12px; -webkit-tap-highlight-color: rgba(0, 0, 0, 0); width: 805px; height: 537px;"> 省略 </div>
```

まだ未熟で何もできていませんが、これは確かに今動いているゲームを示しています。

`<div>`タグで囲まれているのは、enchant.jsが作ったゲームの領域で、ゲームの背景やキャラクターなどは全部この部分に表示されます。

デバッグツール上のこのコードをクリックをすると、Webページ上の該当する部分も青く光って表示されます（光るのはブラウザの基本機能です、enchant.jsやゲームの機能ではありません）。

お疲れ様です！ ここまではよくできました。これからはがんがんゲームの開発に入っていきましょう。

07

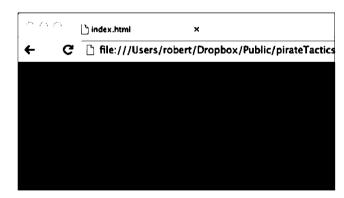
enchant.jsでHello World

この状態だと面白くないので今度は何か文字を表示させてみましょう。第５章でも試したように最初は画面に「Hello World」の文字を出してみましょう。次のようにさきほどのソースコードの`var sceneGameMain = new Scene();`の直後にコードを追加します（これから使うコードはintro → 01にあります）。

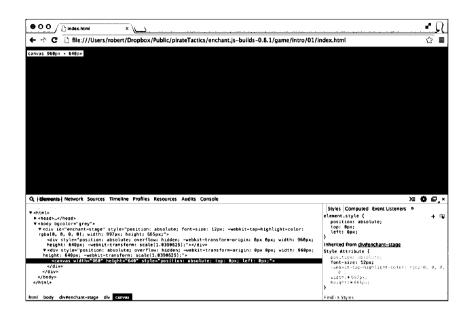
```
var sceneGameMain = new Scene();
var label = new Label("Hello World");

label.x = 200;
label.y = 100;
    sceneGameMain.addChild(label);
game.pushScene(sceneGameMain);
```

ラベルを作ります
ラベルの座標を設定
表示させます

enchant.jsで文字を表示するためのオブジェクトは`Label`です。「Hello World」の文字を作成してから、座標`x=200`と`y=100`を決めて`addChild()`で配置します。

「Hello World」はちゃんと表示されたようです。`label`オブジェクトには次の要素で(文字の色やフォントを)指定することもできます（この本はenchant.jsの解説がメインではありませんので、enchant.jsの機能の詳細についてはWebサイトやほかの書籍の解説を参考にしてください）。

▼ 表7-1 labelオブジェクトに設定できる属性

属性	説明
`color`	文字色の指定
`font`	フォントの指定
`text`	表示するテキスト
`textAlign`	テキストの水平位置の指定

文字は無事に出ましたが、その仕組みをもう少し説明しておきましょう。デバッグツールでゲームを確認してみます。

黒い三角のマークをクリックしていくと、いろいろな要素が表示されます。この中に、「canvas」という要素が追加されていました。画面に何かを表示させると、目には見えませんが、このcanvasという要素がブラウザ上に配置されます。

```
<canvas width="960" height="640" style="position: absolute; top: 0px; left: 0px;"></canvas>
```

07

canvasはenchant.jsの機能ではありません。HTML5に含まれる要素のひとつです。canvasは比較的最近登場した機能で、HTML5以前にはありませんでした。

canvasはグラフィックの描画がとても高速で、今時のほとんどの2Dブラウザゲームはcanvasの恩恵にあずかっています。canvas（とHTML5）のおかげで、ブラウザでちゃんとした2Dゲームが作れるようになってきたのです。

スプライトを表示

「Hello World」をクリアしたところで、canvasという要素の存在が分かりました。せっかく2Dのゲームを作るのですから、canvasの上に絵を出しましょう。ゲームでは画面上に表示しているキャラクターなどの絵を**スプライト**（Sprite）と呼びます。スプライトには画像を画面に表示させ、動かしたり、消したりできる仕組みが用意されています。enchant.jsではスプライトを簡単に利用することができます(これから使うコードは intro → 02にあります)。

ただし、読み込む画像はブラウザの外部に置かれていますので、それをゲームの中に読み込む必要があります。そこでもう一度イベントの仕組みを使ってみましょう（ソースコードの全体は intro → 02 の main.js です)。

```
省略
window.onload = function(){
中略
    var sceneGameMain = new Scene();
    game.preload("../../../resources/pirate00.png");

    game.onload = function(){
中略
    }
    game.start();
}
```

preloadで画像ファイルを読み込む

最初に画像のパス（ファイルの位置）を`preload`に渡します。プリロードとはゲームが始まる前に外部リソース(たとえばファイルや音声)を読み込んでおくための機能です。今は画像1つの読み込みですが、量が増えるとenchant.jsは読み込みの進捗をバーで表示してくれます。スプライトの属性やメソッドのうち主要なものを紹介します。

▼ 表7-2 Spriteの主な属性／メソッド

コンストラクタ	説 明
enchant.Sprite(width, height)	スプライトの大きさ（縦横）を指定する

属 性	説 明
frame	表示するフレームのインデックス番号
height	スプライトの高さ
width	スプライトの幅
image	表示する画像
age	画面に表示されてから経過したフレーム数
parentNode	親オブジェクト
scene	属しているScene
x	x座標
y	y座標

メソッド	説 明
moveBy(x, y)	スプライトの移動（相対指定）
moveTo(x, y)	スプライトの移動（絶対指定）

外部リソースの読み込みが終わるとgame.onloadのイベントが発生します。最初はちょっと不思議に思うかもしれませんが、このように複数のイベントが段階的に呼び出されることはよくあります。たとえば今回はwindow.onloadが終わって、所定の処理をして、次のgame.onloadのイベントが発生するまでのあいだ、ゲームは停止しています。そのあいだに裏ではフレームワークが画像の読み込み作業を行っているのです。

画像の読み込みが完了すると次のようにgame.onloadイベントの処理が実行されます。

省略

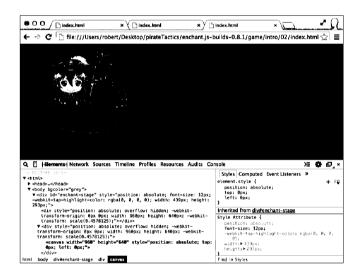
```
game.onload = function(){
    var sprite = new Sprite(256, 256);
    sprite.x = 200;
    sprite.y = 100;

    sprite.image = game.assets["../../../resources/pirate00.png"];
    sceneGameMain.addChild(sprite);
```

スプライトを作成

imageに画像ファイルを当てはめる

```
        game.pushScene(sceneGameMain);
    }
省略
```

最初にSprite()で新しいスプライトを作っています。そして、プリロードした画像をspriteのimageとして設定してから、ゲームのメインシーンに追加します。

この流れは「Hello World」とほとんど同じです。すべてうまく行けばゲームに恐ろしい海賊の姿が現れます。

この本では今から作るゲームに必要なenchant.jsの知識を説明していきます。ただし、enchant.jsには、この本では使わない機能とプラグインもまだまだ沢山ありますのでさらに理解を進めたい人はenchant.jsの解説書などを参考にしてください。

「ゼロからはじめるenchant.js入門」

布留川英一/伏見遼平/田中諒 著、アスキー・メディアワークス、ISBN978-4-0488-6258-5

本章で勉強すること

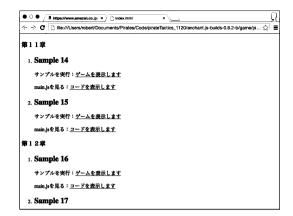
- まずはゲームの舞台となるフィールドを作ってみよう
- 枠、地図、マップの３つのレイヤーを重ねます
- 部品をクラスとして作ることで、プログラミングの見通しを良くしよう
- さらにタッチイベントを作ってみよう
- タッチした箇所の座標を調べて、通行できるかできないかわかる仕組みを作ります
- マップのデータから通行の可否を判断できるデータを生成してみよう

08

開発したいゲームに必要な基礎知識の説明とフレームワークの準備ができましたので、これからは本編のゲーム開発に入りましょう。

まず、ゲームに対してプレイヤーが何かをインプット（操作）をするとアウトプット（結果）が画面に現れるようにしてみましょう。今回のゲームであれば最初はフィールド（海や陸地の背景）を描画し、タ

ッチイベントを処理するようにします。

　これから本書で使うコードはすべてDesktop → enchant.js-builds-0.8.2-b → game → piratesの下の各章のフォルダ内にあります。確認を便利にするためenchant.js-builds-0.8.2-b → game → pirates → index.htmlに各サンプルへのリンクを用意しました。詳しく見たいときはそちらを参照しながら確認してください。

フィールドの構成

　最終的には次のイメージのようなフィールドを完成させましょう。マス目に沿ってアイコンを並べたタイプもありますが、今回は見た目を重視して地図の絵を表示し、その上にマス目を重ねます。ゲームの仕

チームで作るときは

チームで作るためのいろいろな方法があります

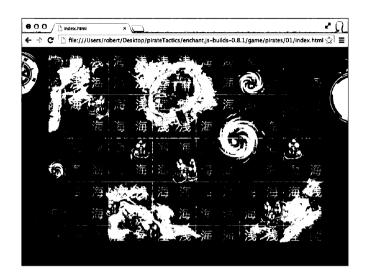
ゲームを作るときは、開発チームや作りたいゲームに合わせて、具体的な開発手順が変わります。今回の本で紹介している開発の手順はおもに一人で作るときに適した順番です。

もしチームで作るなら、参加者ができる限り効率良く開発を進められる体制にする必要があります。たとえば、ゲームのグラフィックを先に進めるなら、最初は仮でもよいのでゲーム画面を描画する部分を作っておき、作業やテストを進めてもらうようにしたほうがよいでしょう。ほかにも、

- ゲームのソースコードを共有して管理するシステム
- 進行状況を把握するためのコミュニケーションツール（メーリングリストなど）
- 問題点などを管理して解決されたかをチェックするシステム（チケットシステムなど）

が必要かもしれません。これらのツールにはさまざまなものがあります。現在は、ソースコード管理ツールGitと組み合わせて利用できるGitHubも人気があります。GitHubに慣れている人も多いので、チームで開発するときは検討してみてもよいでしょう。

GitHub
http://github.com

組み上、内部的にはマス目が存在しますが、このマス目は通常はプレイヤーの目には見えません。フィールド上の駒を動かすときに見えるようになります。

画面全体とフィールドの細かいレイアウトは次のように簡単（かつ必要十分）な仕様書にまとめてあります。

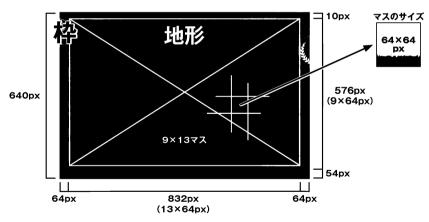
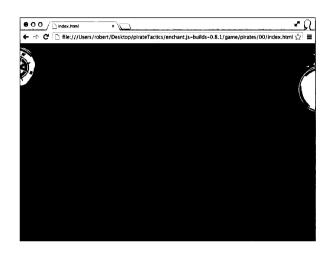



図8-1 ゲーム画面のレイアウト

それぞれのマスは64x64 px（ピクセル）、ゲームフィールドは13x9マスです、あとはフィールドの枠エリアも作って周りに余裕を持たせましょう。特に枠の下の部分にはボタンなどもいろいろ表示できる領域を用意しておきます。右と左の枠はたんに雰囲気作りですが、将来的に自分でゲームを拡張したければ、そのエリアにボタンなどを作ってもよいでしょう。

マップを表示

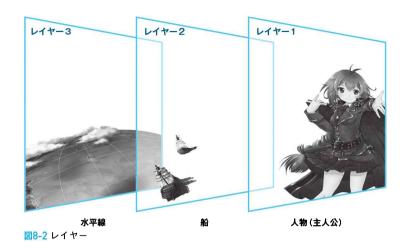
最初にマップをスプライトとして表示します。ここはファイル名が`pirate00.png`から`mapframe.png`に変わっただけで、表示のさせ方は第7章のスプライトのサンプルと同じです。

1枚の絵を貼っただけでもちょっと雰囲気が出てきましたね。でも、この最初のステップは最低限のプロジェクト構成、画像の位置などのチェックになっています。簡単な絵の描画だけでも、裏ではいろいろな仕組みが動いています（新しいゲームを作るときは、このような最初のプロジェクトの段取りでも、けっこうミスがあるものです）。

レイヤー構成

ではこれからは仕様に従い、マップを最終的な形にもっていきましょう。そのために**レイヤー**（Layer）という概念を説明します。

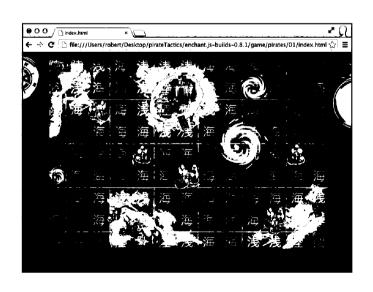
レイヤーはグラフィック系のソフトによく出てくるので知っている方もいるかもしれません。もともとレイヤーという言葉は「重なっている層」を意味しています。たとえば次の図のような画像を作るときに、空／背景／キャラクターと手前のエフェクトをレイヤーで管理することがよくあります。

08

図8-2 レイヤー

今回のゲームのマップは次のような３つのレイヤーで構成したいと思います（コードは pirates → chapter_08 → 01にあります）。

1. 枠のレイヤー
2. 地図のレイヤー
3. ゲームロジック用のマス目のレイヤー

枠と地図のレイヤーは今までと同じスプライトですが、3つ目のマス目のレイヤーはenchant.jsの**マップ機能**を利用しています。まず、枠と背景を重ねてみましょう。

08

main.jsに次のように書いてください。

```
game.onload = function(){
    var sceneGameMain = new Scene();

    // 枠
    var frame = new Sprite(960, 640);
    frame.image = game.assets[mapFrame];
    sceneGameMain.addChild(frame);

    // 背景
    var background = new Sprite(64*13, 64*9);
    background.image = game.assets[mapBackground00];
    background.x = 64;
    background.y = 10;
    sceneGameMain.addChild(background);
```

960×640のスプライトを作成

枠の画像を読み込む

画面の枠を描画する

同様に背景も表示します

気が付いたかもしれませんが、レイヤーを表すコードは特にありません。レイヤーは基本的に考え方であって、命令ではありません。addChildという命令でシーンに要素を加えていくと、それらがレイヤーとして重なって表示されるのです（ただし複数の要素をまとめて管理したい場合もあるので、その場合はenchant.jsのGroupと言う機能をレイヤーのために利用してください）。

次は、マスを配置してみましょう。今までのスプライトと違ってすこし変わった画像ファイルを用意します。

この画像を見てみると、1枚の画像ファイルの中に複数のイメージが含まれています。このような画像は**スプライトシート**（Sprite Sheet）と呼ばれます。このスプライトシートには、いろいろな種類の64x64のマスの画像が含まれています。

enchant.jsに限らず、2Dゲームではこのような複数の画像をまとめた画像を使うことが多くあります。enchant.jsにはMapという機能が備わっているので、これを使えばマス目が簡単に作れます。

```
// マス
var map = new Map(64, 64);
map.x = 64;
map.y = 10;

map.image = game.assets[mapTiles];

var mapDisplayData = [
    [0, 0, 0, 0, 2, 2, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 3, 3, 2, 0, 1, 0, 0, 0],
    [0, 0, 4, 0, 2, 3, 3, 2, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 2, 0, 1, 1, 0, 0, 0],
    [0, 0, 0, 0, 4, 0, 0, 0, 1, 1, 0, 4, 0],
    [1, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 2],
    [0, 0, 0, 3, 3, 2, 0, 0, 0, 0, 4, 2, 2],
    [0, 0, 0, 3, 3, 3, 2, 0, 0, 2, 2, 3, 3],
];

map.loadData(mapDisplayData);
```


new Map(64, 64)のところでタイルのサイズを64×64ドットに指定し、**game.assets[mapTiles]**であらかじめ用意されていた**mapTiles.png**を読み込んでいます。

▼ 表8-1 Mapクラスの主な属性／メソッド

コンストラクタ	説明
enchant.Map(width, height)	タイルの幅と高さを指定する

属性	説明
collisionData	タイルが衝突判定を持つかを表す二元配列
image	Mapで表示するタイルセット画像
tileHeight	Mapのタイルの高さ
tileWidth	Mapのタイルの横幅

メソッド	説明
checkTile(x, y)	ある座標のタイルが何か調べる
hitTest(x, y)	Map上に障害物があるかどうかを判定する
loadData(data)	データを設定する

mapDisplayDataはマスの並べ方を指定しています。数字は64×64に区切られたマス目に付けられた番号です。0から4までの5種類のマスの組み合わせでマップができました。

ほかにサンプルでは、スプライトの 透過度（opacity）を指定しています。opacity=1は不透明、opacity=0は完全に透明です。中間の値を指定することもできます。

Column

なぜスプライトシートを使うのか

スプライトシートは、**マップシステム**あるいは**タイルシステム**とも呼ばれ、2Dゲームでよく使われます。スプライトシートは大きな画像の一部だけを表示することで複数の画像を表示できる方法です。

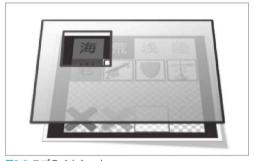



図8-3 スプライトシート

スプライトシートを使う理由のおもなものはパフォーマンスです。画像が大きくても小さくても1つの画像を読み込む時間はあまり変わりませんが、数が増えると時間がかかります。たとえば、16枚の画像を読み込むより、1つのスプライトシートとして読み込めば時間が1/16ですみます。マップのようにたくさんの細かい画像を描画したいときにスプライトシートを使うと、ブラウザやハードウェアの負荷も少なくてすみます。ほかにもアニメーションの連続画像を1つのスプライトシートにまとめることなどがあります。こちらも読み込み時間と描画の負荷を減らせるので、本書でも使ってみることにしましょう。

クラスを使う

これからコードがどんどん増えていきますので、書きやすく／使いやすくするためにマップに関連したロジックやデータを**クラス**としてまとめましょう。

クラスってどんなもの？

クラスをWikipediaで調べてみたら次のような説明になっていました。

> **クラス**（class）
> クラス（class）は、クラスベースのオブジェクト指向においてオブジェクトの設計図にあたるもの。抽象データ型の一つ。クラスから生成したオブジェクトのことをインスタンスという。—wikipedia（http://ja.wikipedia.org/wiki/クラス_(コンピュータ)）

しかし、これではなかなか分かりづらいですね。この本のために、かなり大雑把に説明すると、クラスは「型」のようなものだといえます。
たとえば、船の型を使えば、船のクッキーが簡単にたくさん作れます。これと同じように、プログラミングでクラスを作っておくと同じタイプの部品をたくさん作れるのです（もちろん変化を付けることもできます）。クラスから作られたプログラミングの部品を**オブジェクト**と呼びます。たとえば、1つの船のクラスから複数の船（オブジェクト）が作れますのでいちいちプログラミングする手間が省けるのです。かなり大雑把な説明になりますが、こういう考え方を**オブジェクト指向プログラミング**と呼びます。

プログラミングのスタイルには、手続き型、関数型、エージェント指向、データ指向などいろいろなものがあります。そしてそれぞれに優れた点があります。ただ、ゲームにはモノ（オブジェクト）がたくさんあるため、オブジェクト指向に向いているといえるでしょう。また、AIの分野では**エージェント指向**の影響がありますので、開発をしていると耳にすることがあります。描画周りやオンラインゲームでは**関数型プログラミング**や**データ指向**がよく出てきますので、それぞれを概念だけでも理解しておくと役立つと思います。

では、enchant.jsでのクラスの作り方を説明しましょう。enchant.jsのクラスは、`Class.create(superclass, definition)`という関数で作れます。次の例では、`Bear`というクラスを作っています。

08

継承元のクラス
自作クラスBearの定義
クラスの初期化関数（コンストラクタ）
スプライトの初期化
画像の指定

```
Bear = Class.create(Sprite, {
  initialize:function(){
    Sprite.call(this,32,32);
    this.image = game.assets['fig1.png'];
    game.rootScene.addChild(this);
  }
});
```

オブジェクトが作られたときに自動で実行する処理は、`initialize:function(){`のところに書いておきます。これを**コンストラクタ**と呼びます。

クラスを定義すると、今度はもっと簡単にキャラクターを出せるようになります。

```
bear = new Bear();
```

これだけです。ここでは、Spriteクラスを元にして自作のBearクラスを作りました。これを**継承**と呼びます。継承すると、Spriteの機能をすべてもったBearクラスが簡単に作れます。継承を使わない場合は、次のように書きます。

継承元がない

```
Class.create({
    initialize:function(){
        this.x=10;
    });
```

また、クラスには**イベントリスナー**を書いておくこともできます。

自作クラスBearの定義
クラスの初期化関数（コンストラクタ）
スプライトの初期化
画像の指定

```
Bear = Class.create(Sprite, {
    initialize:function(){
        Sprite.call(this,32,32);
        this.image = game.assets['fig1.png'];
        game.rootScene.addChild(this);
    },
```

08

```
    onenterframe:function(){
        this.x+=2;
    },
    ontouchend:function(){
        this.y+=10;
    }
});
```

enterframeイベントのイベントリスナー

touchendイベントのイベントリスナー

こうするとイベントリスナーをクラスごとにまとめて書いておけるのでコードがすっきりします。一度作ったクラスは、いくつでもコピーできます。

```
bear1 = new Bear();
bear1.x=100;

bear2 = new Bear();
bear2.x=150;
```

Bearからbear1を作ります

Bearからbear2を作ります

bear1と**bear2**にはそれぞれ**x**というプロパティがありますが、これらは別のものとして扱われます。クラスをうまく活用することで効率的な開発が可能になります。

GameMapのクラスを定義しよう

では今作っているゲームの開発に戻りましょう。クラスの説明を見て気が付いた人もいると思いますが、今までに使った**Label**、**Sprite**や **Map**はどれもクラスです。こちらのクラス、そしてこれら以外の便利なクラスはあらかじめenchant.jsのフレームワークに入っています。しかし、ゲームに必要なクラスは自分で作らなければいけません。たとえば、「地図」「船」「技」などです。ではこれから地図のクラスを作ってみましょう。main.jsに**Map**クラスを定義します（コードは pirates → chapter_08 → 02にあります）。

```
/**
 * Map クラス
 */
```

08

```
var GameMap = Class.create({
    initialize: function(scene, mapData) {
        // 枠
        var frame = new Sprite(960, 640);
        frame.image = game.assets[mapFrame];
        scene.addChild(frame);

        // 背景
        var background = new Sprite(64*13, 64*9);
        background.image = game.assets[mapBackground00];
        background.x = 64;
        background.y = 10;
        scene.addChild(background);

        // マス
        var tiles = new Map(64, 64);
        tiles.image = game.assets[mapTiles];
        tiles.x = 64;
        tiles.y = 10;
        tiles.loadData(mapData);
        tiles.opacity = 0.5;

        scene.addChild(tiles);
    },
});
```

枠サイズは960×640ドット

背景は縦13マス×横9マスぶん

Mapを使ってマス目を準備

タイルの透明性

新しく作ったGameMapのクラスは次のように使えます。

```
// マスのデータ
var mapDisplayData = [
    中略
];

var map = new GameMap(sceneGameMain, mapDisplayData);
```

追加するシーン

マスのデータを指定

タッチイベントを作ってみよう

マップのデータをクラスを使って書いてまとめたところで、次はマップにタッチできるようにしましょう。次のように単純なタッチイベントを書いています（コードは pirates → chapter_08 → 03にあります)。

```
/**
 * Map クラス
 */
var GameMap = Class.create({
    initialize: function(scene, mapData) {
        中略

        // マス
        var tiles = new Map(64, 64);
        中略

        tiles.touchEnabled = true;
        tiles.addEventListener(enchant.Event.TOUCH_END, function(params){
            alert("タッチ x:"+params.x +", y:"+params.y);
        });
    },
});
```

コンストラクタを定義

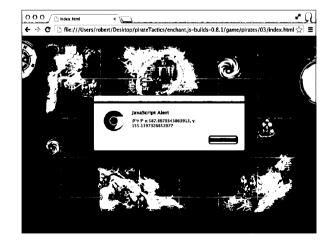
タッチを可能にする

自身に対するタッチの終了イベント

タッチのあった箇所のX座標とY座標を示す

まず、**tiles.touchEnabled = true**でタッチが可能な状態に設定します。イベント処理は基本的に**addEventListener([イベント名], [ハンドラ])**で書いていきます。ここでは、enchant.jsの**Event**クラスが持つ、**TOUCH_END**というイベントに対し、**alert**関数を実行しています。

この状態でマップをタッチするとポップアップが出てきます。

08

ポップアップの中にタッチした箇所の座標が示されます。enchant.jsのEventクラスのイベントはTOUCH_ENDイベント以外にもいろいろなものがあります。ゲームに使えそうなものをいくつか紹介しておきましょう。

▼ 表8-2 Eventクラスのイベント（ユーザーの操作に関わるもの）

イベント	説明
enchant.Event.A_BUTTON_DOWN	aボタンが押された
enchant.Event.A_BUTTON_UP	aボタンが離された
enchant.Event.B_BUTTON_DOWN	bボタンが押された
enchant.Event.B_BUTTON_UP	bボタンが離された
enchant.Event.UP_BUTTON_DOWN	upボタンが押された
enchant.Event.UP_BUTTON_UP	upボタンが離された
enchant.Event.DOWN_BUTTON_DOWN	downボタンが押された
enchant.Event.DOWN_BUTTON_UP	downボタンが離された
enchant.Event.RIGHT_BUTTON_DOWN	rightボタンが押された
enchant.Event.RIGHT_BUTTON_UP	rightボタンが離された
enchant.Event.LEFT_BUTTON_DOWN	leftボタンが押された
enchant.Event.LEFT_BUTTON_UP	leftボタンが離された
enchant.Event.INPUT_START	ボタン入力が始まった
enchant.Event.INPUT_CHANGE	ボタン入力が変化した
enchant.Event.INPUT_END	ボタン入力が終了した
enchant.Event.TOUCH_END	自身に対するタッチが終了した
enchant.Event.TOUCH_MOVE	自身に対するタッチが移動した
enchant.Event.TOUCH_START	自身に対するタッチが始まった

ほかにもたくさんのイベントが定義されています。詳しく知りたい場合はenchant.jsのリファレンスを参考にしてください。

enchant.jsのイベント
http://wise9.github.io/enchant.js/doc/core/ja/symbols/enchant.Event.html

通行できるか／できないか？

これでメッセージは出せましたが、ゲームのためには座標ではなくどのマスをタッチしたのか知りたいところですね。enchant.jsでは、`Map`クラスに付いている機能で座標からマップの情報を得ることができます（コードはpirates → chapter_08 → 04）。

判定用データの作り方

まず、プログラムで地形を扱い易いようにマップの`id`に対して名前を対応付けておきましょう。

```
var tileTypes = {
    umi:  {id:0, name:"umi"},
    arai: {id:1, name:"arai"},
    asai: {id:2, name:"asai"},
    riku: {id:3, name:"riku"},
    iwa:  {id:4, name:"iwa"},
};
```

id:0→海
id:1→荒い海
id:2→浅い海
id:3→陸
id:4→岩

`umi`、`arai`、`asai`、`riku`、`iwa`の5種類のマスを定義しました。

次に、マスごとの通行可能／通行不可能のマスを判定するためのデータを作ります。こんな配列を`mapDisplayData`から自動で作れるようにプログラムを書きます。

```
var mapCollisionData = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1],
    [0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
];
```

判定データは0と1だけを使用しています。0は通行可能、1は通行不可の意味になります。

これくらい小さなマップでわざわざ判定用のデータを持つのは本当は大袈裟なのですが、ゲームが大きくなると描画マップと判定マップを別に管理することは珍しくありません。勉強の意味も兼ねて今回はこのように書いてみましょう。

今回の地形だと陸と岩は通行不可なので、それを元に判定データを作成します。

```
// マップの大きさを保存
this.mapHeight = mapData.length;
this.mapWidth  = mapData[0].length;

// 元のマップデータから陸や岩のcollisionデータを生成
var mapCollisionData = [];
for(var j=0; j < this.mapHeight; j++) {      ← マップの縦のぶんだけループする
    mapCollisionData[j] = []
    for(var i=0; i < this.mapWidth; i++) {   ← マップの横のぶんだけループする
        if (mapData[j][i] == tileTypes.riku.id || mapData[j][i] ==
        tileTypes.iwa.id) {                  ← もしも「陸」または「岩」だったら
            mapCollisionData[j].push(1);     ← 通行不可
        } else {
            mapCollisionData[j].push(0);     ← それ以外は通行可
        }
    }
}
this.tiles.collisionData = mapCollisionData
```

マップデータを調べて、idがrikuあるいはiwaのものであれば1を、それ以外は0を配列に代入していきます。これでデータの準備は整いました。あとはタッチの挙動を作り込みます。

座標を変換する

ここまでのサンプルをよく見るとタッチイベントで表示された座標は画面の左上の角からの距離になっています。マップはゲーム画面の上から10ピクセル、左から64ピクセルの位置にあるので、そのままマップの判定に渡すとズレが生じます。これはゲームを作るときによく直面する問題です（これから使うコードは pirates → chapter_08 → 04bにあります)。

どの世界でも位置を表現する座標は多数（無限）にあります。たとえば、現在地を知るにはGPSによる座標もありますが、町の郵便番号という座標もありますね。ほかにも広い宇宙全体から見た世界共通の座標もあるかもしれません。

世界共通の座標は、ゲームでいえば**ワールド座標**もしくく**絶対座標**です。enchant.jsのタッチイベントで取れる位置情報もこのワールド座標（ゲーム画面の左上端からの位置）で示されています。

これとは別に、ゲーム画面上のどこかの位置を基準にした別の座標も考えられます。普段の生活でもいつもGPS上の座標を気にしているわけではなく、いま自分のいる位置から「10m向う、5m左」というような考え方をしますね。これはゲームの用語でいうと**ローカル座標**もしくは**相対座標**です。

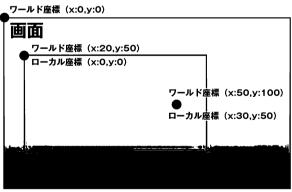



図8-4 ワールド座標とローカル座標

今回のタッチイベントの位置もワールド座標からローカル座標に変更しましょう。数学的に座標の変換にはさまざまな理論がありますが、今回は普通の四角い2Dスペースですから、たんに目的のワールド座標からマップのある位置の座標を引けば目的のローカル座標か計算できます。クラスのメソッドにするとこのようになります。

```
toLocalSpace:function(x,y) {
    var localX = x -this.tiles.x;
    var localY = y -this.tiles.y;
    return {x:localX, y:localY}
},
```

ワールド座標からtilesの座標をマイナス

ここまでのノウハウを組み合わせるとタッチイベントから地形データを正しく読み取れるようになります。

```
ontouchend:function(params) {
    // collisionデータを利用して判定をする
    if (this.hitTest(params.x, params.y) == true) {
        alert("通れない")
    } else {
        alert("通れる")
    }
},
```

hitTest()関数に座標を入れると、通れるか通れないかが判定されますので、メッセージを表示するようにしました。

ここまでで、タッチイベントから地形を逆引き出来るようになりましたので、次はそれを利用してキャラクターの船を動かせるようにしてみましょう。

本章で勉強すること

- ゲームの「駒」に相当する船のユニットを作ろう
- スプライトを拡張して作ることにします
- enchant.jsの機能を利用して船にアニメーションを加えよう
- 船の移動力のぶんだけ動かす仕組みを作ります
- マップ内で距離を測る3つの方法に関する知識を付けましょう
- クリックすると動ける範囲が表示される仕組みも作り込もう

船クラスを作る

地図のタッチイベントができたので次は船のクラスを作って動かせるようにしたいと思います。

船クラスは基本的にSpriteクラスを拡張して作ります。あるクラスを元にして新しいクラスを作ることをオブジェクト指向の用語で**継承**（inheritance）と呼びます。ちょっと単純化しすぎかもしれませんが、継承は元のクラスの機能を引き継いで何かをプラスすることだと思ってください。`Sprite`クラスを元にした`Fune`クラスのコードは次のようになります（これから使うコードは pirates → chapter_09 → 05 にあります）。

```
var Fune = Class.create(Sprite, {
    initialize: function(scene) {
        Sprite.call(this, 96, 96);

        this.image = game.assets[shipsSpriteSheet];
    }
});
```


継承すると`Fune`は`Sprite`のメソッドやデータに簡単にアクセスできます。

enchant.jsは基本的にJavaScriptの継承システムに沿っていますので、コンストラクタに新しい機能を加えたいときは、例示したコードのように`Sprite.call(this, 96, 96);`としてまず`Sprite`クラスのコンストラクタを実行する必要があります。これで自分のオブジェクト（`this`）に継承元のクラス（`Sprite`）の初期化ロジック（コンストラクタ）が反映されます。この手順を行う理由を理解するにはJavaScript言語のプロトタイプ機能についての知識が必要です。興味があれば調べてみていただきたいですが、今回のようにゲームを作るためだけであれば「継承するときは、この手順を忘れずに行う」と覚えてもらっても十分です。

継承と集約

ここで、ちょっと気になった人がいるかもしれません。前の章で作成したGameMapクラスはMapクラスを継承しないでもよかったのでしょうか？ GameMapクラスにはたしかにMapの機能があります。しかし、Map以外の枠や背景などにはSpriteの要素も含まれていました。このように複数のクラスの機能を併せ持つ場合は**集約**（aggrigation）あるいは**複合**（composition）と呼ばれています。
クラスを作るとき継承にするか集約するかには、さまざまな考え方があります。これはちょっと哲学的でもあって、答えもひとつではありません。
実際には「GameMapは継承でもいいのか？」、「FuneはSpriteを集約してもいいではないか？」という議論もありますが、今回は両方のやり方を紹介するため、GameMapは集約、Funeは継承で進めたいと思います。

マスの空間

このシミュレーションゲームはマスを使っていますので、実際に船をマップ上で動かすときに座標（x,y）を単位に指定するのは不便です。そのため別の単位を決めましょう。今回はマス目単位の軸（i軸とj軸）を使うことにします。

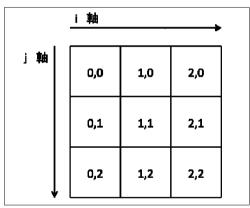



図9-1 マス目のi軸とj軸

GameMapに次のような単位の変換メソッドを追加しています（詳細はサンプルコードで確認してください）。

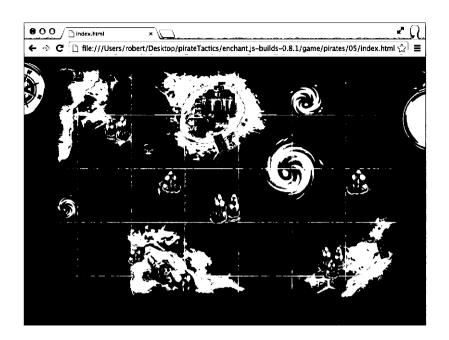
- **toLocalSpace:function(worldX, worldY)：ワールド座標を与えるとローカル座標（localX, localY）を返す**
- **toWorldSpace:function(localX, localY)：ローカル座標を与えるとワールド座標**

（worldX, worldY）を返す
・getMapTileAtPosition:function(localX, localY)：ローカル座標を与えるとマス目（i, j）を返す
・getMapPositionAtTile:function(i,j)：マス目を与えるとローカル座標（localX, localY）を返す

これを利用してfuneオブジェクトを配置するための関数を追加しました

```
positonObject: function(object, i, j) {

    var postion = this.getMapPositionAtTile(i, j);
    var worldPosition = this.toWorldSpace(postion.localX,postion.localY);

    object.x = worldPosition.x;
    object.y = worldPosition.y;
},

positonFune: function(fune, i, j) {
    this.positonObject(fune, i, j);
},
```


最後は船をマップに置きましょう。game.onloadに次の2行を追加して船をi:3,j:3に配置します。

```
var fune = new Fune();
map.addChild(fune);
map.positonFune(fune, 3, 3);
```

船にアニメーションを付けよう

船をタッチで動かす前に、ちょっとゲームらしくするためにユラユラと船がアニメーションするようにしましょう（コードはchapter_08 → 06にあります）。

アニメーションさせるには一定時間で画面を更新させます。1秒に何回に画面を更新をするかという単位はFPS（Frames Per Second）です。enchant.jsでFPSを設定するのは簡単です。`game.fps = 30;`で30fpsを指定しました。

```
window.onload = function(){
    var game = new Core(960, 640);
    game.fps = 30;
    後略
```

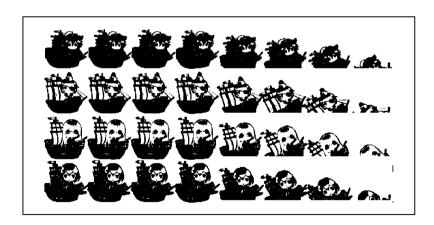
FPSを指定

09

理想のFPSは？

ゲームの世界で理想的なのは60fpsです。今の時代のテレビは最大60fpsの再生能力がありますので、それ以上では無駄が出ることになります。
60fpsということは、1秒=1000ミリ秒として単純計算すると16ミリ秒以内に画面更新を終わらせなければいけません。PS3やXbox 360などの家庭用ゲームのほとんどは24 ～ 30fpsでした。その理由は家庭用ゲーム機の能力で16ミリ秒以内に3D描画、物理演算、AIなどを処理するのは難しかったからです。
意外なことに初代PlayStationやそれ以前の世代のゲーム機には60fpsゲームがけっこうありました。逆に言うと前世代の家庭用ゲームはかなり無茶をしていたということです。最近になり、PS4やXbox Oneなどのハードでは60fpsのゲームが増えてきました。
いま作っている海賊ゲームの場合、デスクトップのブラウザなら60fpsは出るかもしれませんが、JavaScriptは少々重い言語です（最新ブラウザではそんなことはないですが、ちょっと旧い世代だと処理が遅くなることもありました）。また、スマートフォンのブラウザはさまざまですので、30fpsにしておくほうが無難です。アクション性の少ないゲームなので30fpsでも十分ともいえるでしょう。

船の画像にはスプライトシートを使いましょう。いろいろな船が描いてありますが、今はひとつの船をアニメーションをさせます。

動かすにはenchant.jsのスプライトシートを利用したアニメーション機能を利用します。frameメソッドにシートの番号を与えるだけで簡単です（コードは chapter_08 → 06bにあります）。

```
var Fune = Class.create(Sprite, {
    initialize: function(scene) {
        Sprite.call(this, 96, 96);
        this.image = game.assets[shipsSpriteSheet];
        this.frame = [0, 1, 2, 3];
    }
});
```

表示させるコマを順に記述する

サンプルを動かすとたしかに船はアニメーションをしていますが、ちょっと動きが速くて変ですね。アニメーションの順番を変えましょう。

```
initialize: function(scene) {
    中略
    this.frame = [0, 0, 0, 0, 1, 1, 1, 2, 2, 1, 1, 0, 0, 0, 0, 3, 3, 3];
}
```

フレームを適切なアニメーションの速度に合わせました。

船を地図上で動かす

GameMapに船を動かせるように設定できるメソッドを追加して、game.onloadで設定します（コードは chapter_09 → 07にあります）。

```
var fune = new Fune();
map.positonFune(fune, 3, 3);
map.setActiveFune(fune);
```

動かせる船を指定するメソッド

funeというオブジェクトを作ったら、3,3の位置に配置して、アクティブにしています。

あとはGameMapのタッチイベントの処理に、船を移動する処理を追加します。

```
ontouchend:function(params) {
    var localPosition = this.toLocalSpace(params.x, params.y);

    var tileData = this.tiles.checkTile(localPosition.x, localPosition.y);
    var tileInfo = this.getTileInfo(tileData);

    if (this.tiles.hitTest(localPosition.x, localPosition.y) == true) {
        alert("通れない、"+tileInfo.name);
    } else {
        var tile = this.getMapTileAtPosition(localPosition.x,
        localPosition.y);
        this.positonFune(this.activeFune, tile.i, tile.j);
    }
},
```

動かせないマスであればメッセージを表示

動かせるときはクリックした位置からタイルの位置を取得

指定の位置に移動させる

試してみると船がタッチした場所に動きます。また、岩や陸に動かそうとすると、「通れない」というメッセージも出ます。しかし、まだ自由すぎるかもしれませんね。どこをタッチをしても（岩や陸以外なら)そこに移動できます。本来は決まった移動パラメータの範囲に移動を制限する必要あります。そこで、Funeクラスに移動力のパラメータ（movement）を追加しましょう。

```
var Fune = Class.create(Sprite, {
    initialize: function(scene) {
        Sprite.call(this, 96, 96);
        中略
        this.stats = {
            movement: 3,
        };
     },

    getMovement() {
        return this.stats.movement;
    },
});
```

移動力を表すプロパティを追加

移動を取得するメソッド

次に、オブジェクトのi，jの位置を記憶するためmap.positionObjectをすこし変更します。

```
positonObject: function(object, i, j) {
    var postion = this.getMapPositionAtTile(i, j);
    var worldPosition = this.toWorldSpace(postion.localX, postion.localY);

    object.x = worldPosition.x;
    object.y = worldPosition.y;

    object.i = i;
    object.j = j;
},
```

iおよびjを保存

objectのx,yプロパティにはワールド座標が入っています。これで船とタッチしたマスのあいだの距離が計算できるはずです。あとは距離に応じて動かせる範囲を制御する仕組みをタッチイベントの処理内に加えることにしましょう。

```
ontouchend:function(params) {
    var localPosition = this.toLocalSpace(params.x, params.y);

    var tileData = this.tiles.checkTile(localPosition.x, localPosition.y);
    var tileInfo = this.getTileInfo(tileData);

    if (this.tiles.hitTest(localPosition.x, localPosition.y) == true) {
        alert("通れない、"+tileInfo.name);
    } else {
        var tile = this.getMapTileAtPosition(localPosition.x,
        localPosition.y);

        if (this.mapGetDistance(this.activeFune.i, this.activeFune.j,
        tile.i, tile.j) <= this.activeFune.getMovement())
        {
            this.positonFune(this.activeFune, tile.i, tile.j);
        }
    }
},
```

距離を計算するメソッド(まだ作っていません)

距離が移動力以下ならば

移動を実行します

だいたいできたようですが、距離を計算するmapGetDistanceメソッドはまだ作っていません。この部分の作り方を説明します。

ゲームの中の距離を測る

ゲームにおける距離の測り方にはさまざまなものがあります。特にマスを使うゲームの空間では距離は現実世界の距離の計算と異なります。ここではマスを使うゲームの代表的な距離の計算方法を3つ紹介したいと思います。

ユークリッド距離（Euclides Distance）

一番現実世界に近い距離の計算の仕方です。マス目の中心同士の直線距離を測ります。

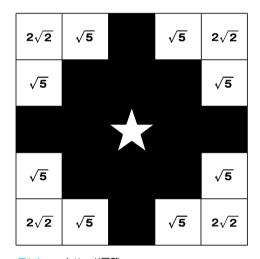



図9-2 ユークリッド距離

これをJavaScriptで計算するコードは次のとおりです。

```
getEuclideanDistance: function(startI, startJ, endI, endJ) {
    var distanceSq = Math.pow(startI -endI, 2) +Math.pow(startJ -endJ, 2);
    var distance   = Math.sqrt(distanceSq);
    return distance;
},
```

Iの距離の2乗とJの距離の2乗を足す

平方根を求める

ユークリッド距離には問題が2つあります。ひとつ目は距離が整数になるとは限らないことです。小数点以下になることもあります。

ふたつ目は計算に平方根が必要なことです。平方根を計算するのはPCでもそれなりに時間がかかります。1回くらいなら余裕ですが、毎フレームあたり数百～数千件を計算するとなるとゲームが処理落ちする（処理が追いつかなくなる）可能性もあります。

$\sqrt{2}$ ＝1.414213562...
$2\sqrt{2}$ ＝2.828427125...
$\sqrt{5}$ ＝2.236067977...

チェビシェフ距離（Chebyshev Distance）

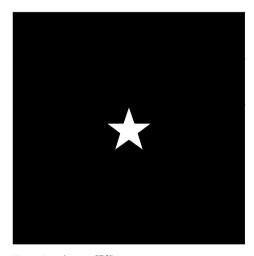



図9-3 チェビシェフ距離

チェビシェフ式は中心のマスを囲むように距離が広がっていく方法です。一番計算しやすく、直観的にも理解しやすい方式です。そのおかげで将棋やチェスのような戦略ボードゲームではよく使われています。これをJavaScriptで計算するコードは次のようになります。

```
getChebyshevDistance: function(startI, startJ, endI, endJ) {
    var distance = Math.max(Math.abs(startI -endI), Math.abs(startJ -endJ));
    return distance;
},
```

Iの距離またはJの距離のいずれか大きな方を返す

09

マンハッタン距離（Manhattan Distance）

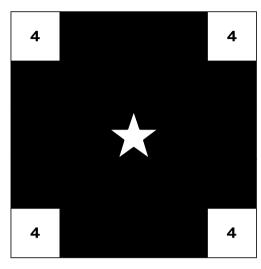



図9-4 マンハッタン距離

マンハッタン距離の由来はニューヨークのマンハッタンを移動するのと似ているからです。マンハッタンの街は道路でほとんど正方形に区切られています。公園でもなければ通常は斜めに移動できません。マスを使用するゲームのほとんどはマンハッタン距離を移動計算に使っています。マンハッタン距離をJavaScriptで計算するコードは次のようになります。

```
getManhattanDistance: function(startI, startJ, endI, endJ) {
    var distance = Math.abs(startI -endI) +Math.abs(startJ -endJ);
    return distance;
},
```

Math.absは絶対値を返します
Iの距離とJの距離の絶対値を足します

今回の海賊ゲームでも移動にはマンハッタン距離を使いましょう。

「円形」は「丸」とは限らない？

学校で
あまり教えない
数学にも
面白いことが
いっぱいありますね！

数学の世界では「中心から一定の距離になる点の集まり」が「円形」とされています。ということは、距離を測る方式によって円形の形状は変わるということになります。たとえば、ユークリッド距離、チェビシェフ距離、マンハッタン距離による円形は次の図のようになります。

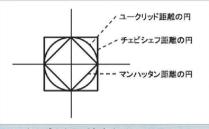
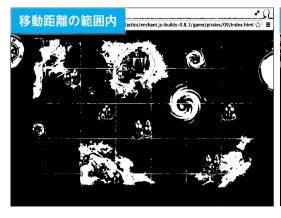
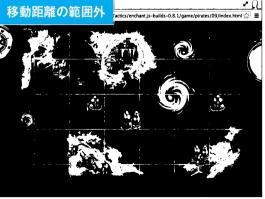



図9-5 さまざまな円が存在する

このように、「丸」の形にはなっていなくても、「円」といえる場合もあります。

ここまでを実装したサンプルはchapter_09 → 08にあります。試してみると船はマンハッタン距離で3以内だけ移動できるようになっていると思います。

タッチイベントを表現

次は、タッチ（入力）を目で見えるように（出力）しましょう。今回のゲームのモチーフは海賊なのでタッチしたところに宝の地図のような×マークを描画しましょう。船が動ける範囲内なら×を赤く、範囲外なら×を半透明な灰色にします。

今回もスプライトシートを使用してフレーム0（赤）とフレーム1（灰色）を切り替えます。タッチのほかにマップにドラッグしたときも×を表示したいので、マップのontouchupdateのイベントハンドラを書いてみます（chapter_09 → 09のサンプル）。

```
ontouchupdate: function(params) {
    var localPosition = this.toLocalSpace(params.x, params.y);
    var tile = this.getMapTileAtPosition(localPosition.x, localPosition.y);
    if (this.outOfBorders(tile.i, tile.j)) {
        return
    }

    if (this.mapMarker == undefined) {
        this.mapMarker = new Sprite(64, 64);
        this.mapMarker.image = game.assets[mapUI];
        this.positonObject(this.mapMarker, tile.i, tile.j);
        this.overLayer.addChild(this.mapMarker);
    } else {

        this.positonObject(this.mapMarker, tile.i, tile.j);
    }

    if (this.tiles.hitTest(localPosition.x, localPosition.y) == true) {
        this.mapMarker.frame = 1;
    } else {
        if (this.getManhattanDistance(this.activeFune.i, this.activeFune.j,
        tile.i, tile.j) <= this.activeFune.getMovement()) {
            this.mapMarker.frame = 0;
        } else {
            this.mapMarker.frame = 1;
        }
    }
}
```

マップ以外のマスのない場所は何もしないのでreturnで中断

×マークのオブジェクトがない場合、×のスプライトオブジェクトを作る

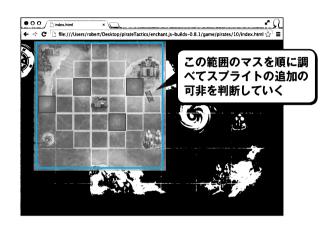
×マークのオブジェクトがあるときは、それを移動する

岩や陸の上には動けないので距離に関係なく灰色マークにする

距離範囲内は×マークを赤く

距離範囲外は×マークを灰色に

動ける範囲を表示しよう

×マークだけでもプレイヤーは触りながら動ける範囲を理解できますが、あまり親切ではないので最初から動ける範囲をマップ上に見えるようにしましょう（chapter_09 → 10のサンプル)。

```
drawMovementRange: function() {
    console.log("update drawMovementRange")

    if (this.areaRangeLayer) {
        this.underLayer.removeChild(this.areaRangeLayer);
        delete this.areaRangeLayer;
    }
```

移動範囲を示すレイヤーがあったら消す

```
    this.areaRangeLayer = new Group();
    this.underLayer.addChild(this.areaRangeLayer);
```

現在地を中心にした四角形の範囲を左下から右上の順に調べていく

```
    for (var rangeI = -this.activeFune.getMovement(); rangeI <= this.
    activeFune.getMovement(); rangeI++) {
        var targetI = this.activeFune.i +rangeI;
        for (var rangeJ = -this.activeFune.getMovement(); rangeJ <= this.
        activeFune.getMovement(); rangeJ++) {
            var targetJ = this.activeFune.j +rangeJ;
```

outOfBordersでマップ内であることを確認

```
            if (!this.outOfBorders(targetI, targetJ)) {
```

09

```
                if (this.getManhattanDistance(this.activeFune.i,
                    this.activeFune.j, targetI, targetJ) <=
                    this.activeFune.getMovement()) {
                        var areaSprite = new Sprite(64, 64);
                        areaSprite.touchEnabled = false;
                        areaSprite.image = game.assets[mapUI];
                        var position = this.getMapPositionAtTile(targetI,
                        targetJ);
                        if (this.tiles.hitTest(position.localX, position.
                        localY) == true) {
                            areaSprite.frame = 3;
                            } else {
                            areaSprite.frame = 2;
                        }
                        this.positonObject(areaSprite, targetI, targetJ);
                        this.areaRangeLayer.addChild(areaSprite);
                    }
                }
            }
        }
    },
```


　サンプルでしばらく遊んでみると分かると思うのですが、現在は移動距離の範囲内なら船はどこにでも行けます。島や岩の向こうにも移動できてしまいます。原因は移動の経路を計算していないからですが、移動を正確に決めるのにはまだ努力が必要です。今のところ、この仕様でも致命的ではないので、第15章まではそのままにしておきましょう。

　これで船と移動の部分の作り込みはできました。それなりにゲームに見えますが、まだルールが足りませんので、次の章でゲームのルールの実装に挑戦しましょう。

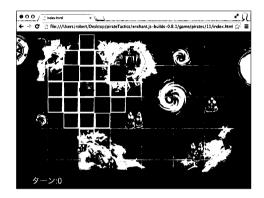
本章で勉強すること

- ゲームにターンを設けることで、ゲームに時間の概念が生まれる
- ゲームの進行を管理するGameManagerクラスを作りましょう
- 複数のプレイヤーとそれに属するユニット、ターンの進行を管理しよう
- クラスを使うことで、プレイヤーやユニットを容易に増やすことができる
- 2人目のプレイヤーを作り、最終的にユニットをそれぞれ4つに増やそう
- TurnUIで現在何ターン目であるか確認できるようにしておこう

前の章ではマップ上に船を表示して、動かすことができるようになりました。ゲームを作る際はここまでがひとつの大きなチェックポイントになります。ゲームのルールはまだ実装できていませんが、基本的な入力と表示はできるようになっています。そこで、次は今作っているゲームにルールを追加していくこ

とにしましょう。

　この本の海賊ゲームはターン制のゲームですので、最初に導入したい機能はターンのシステムと管理です。これがあればゲームに「時間」という概念が生まれます。

ゲーム管理クラス

　ゲームのターンを管理するGameManagerクラスのコードは次のような形になります（サンプルは、chapter_10 → 11にあります）。

```
var GameManager = Class.create({
    initialize: function() {
        this.playerList = [];
        this.turnCounter = 0;
    },

    addPlayer: function(player) {中略},

    setMap: function(map) {中略},

    setTurnUI: function(ui) {中略},

    setStartPositions: function(startPositions) {中略},

    getActivePlayer: function() {中略},
```

- GameManagerクラスを作る
- 当初はプレイヤーもなく、ターンは0
- プレイヤーを追加するメソッド
- マップを追加するメソッド
- ターン表示の追加メソッド
- 初期位置の設定メソッド
- 現在誰のターンかを返すメソッド

```
    beginGame: function() {
        var player = this.getActivePlayer();
        for (var funeIndex = 0; funeIndex < player.getFuneCount();
        funeIndex++) {
            var fune = player.getFune(funeIndex)    船を初期位置に配置
            this.map.addChild(fune);
            var startPosition = this.startPositions.player1[funeIndex]
            this.map.positionFune(fune, startPosition.i, startPosition.j);
        }

        this.startTurn();
    },

    startTurn: function() {    ターンの初期処理

        var player = this.getActivePlayer();
        player.setActive(true);

        this.updateTurn();
    },

    updateTurn: function() {    ターンの更新処理

        this.map.setActiveFune(this.getActivePlayer().getActiveFune());

        this.map.drawMovementRange();    クリックしている船の移動範囲を表示

        this.turnUI.updateTurn(this.turnCounter);
    },

    endTurn: function() {    ターンの終了処理

        var player = this.getActivePlayer();
        player.setActive(false);

                                現在のプレイヤーを非アクティブに
        this.turnCounter++;     してターンカウンターを１つ増やし
        this.startTurn();       次のターンを開始
    },
})
```

管理クラスは、複数のプレイヤーを管理するようになっており、さらにプレイヤーは複数の船のオブジェクトを所有しています。

図10-1 管理クラスはプレイヤーを持っている、プレイヤーは船を持っている

GameManagerクラスにはプレイヤー、マップ、ユニットの開始位置を設定する関数が用意されています。作ったゲーム管理クラスにマップや船を設定します。そして初期化が終わったら**beginGame**を実行します。**beginGame**はターンを開始する**startTurn**を呼び出し、さらに各ターンの処理を行う**updateTurn**を呼び出します。**endTurn**メソッドが呼ばれるとターンの終了処理が実行されますが、今のところゲームを終わらせる処理は書いていないので、交互にユニットをずっと動かし続けられます。

```
game.onload = function(){
    var sceneGameMain = new Scene();

    // ゲームロジックの管理
    var manager = new GameManager();

    // マスのデータ
    var mapDisplayData = [
        中略
    ];

    var map = new GameMap(sceneGameMain, mapDisplayData);
    manager.setMap(map);

    // プレイヤー1
    var player1 = new GamePlayer();    ◀ プレイヤーの作成と船の追加
    manager.addPlayer(player1);

    // 船をプレイヤーに追加
    var fune = new Fune();
    player1.addFune(fune);
```

```
    // 船の初期の位置
    var startPositions = {
        player1: [
            {i:3, j:3}
        ],
    }
    manager.setStartPositions(startPositions);

    // ゲームにシーンを追加
    game.pushScene(sceneGameMain);

    // ゲームのロジック開始
    manager.beginGame();   ← ここでゲームが開始されます
};
```

これだけでターンの管理はできていますが、今はどのターンであるかを表示したいので**TurnUI**と言うクラスを作り、それも**GameManager**に管理をさせることにしました。

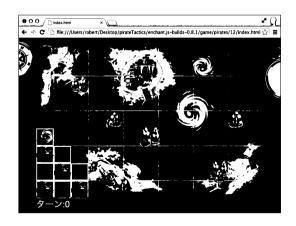
```
var TurnUI = Class.create(Label, {   ← ターンの表示を行うTurnUIクラス
    initialize: function(scene) {
        Label.call(this);
        scene.addChild(this);
        this.x = 64;
        this.y = 640 -50;
        this.font = "32px 'MS ゴシック', arial, sans-serif";   ← CSS式に文字スタイルを設定
        this.color = "rgba(20, 20, 255, 1.0)"
    },

    updateTurn: function(turn) {
        this.text = "ターン:"+turn;
    },
})

game.onload = function(){
        中略
    var map = new GameMap(sceneGameMain, mapDisplayData);
    manager.setMap(map);
```

```
    var turnUI = new TurnUI(sceneGameMain);
    manager.setTurnUI(turnUI);

    // プレイヤー1
    var player1 = new GamePlayer();
    manager.addPlayer(player1);
    中略
};
```

船を増やしてみる

GDDによると船は4種類あるので、船を増やしましょう。クラスにしておけば、同じコードを繰り返し書く必要はありません。

```
// プレイヤー1に船を4つあげよう
for (var i=0; i <4; ++i) {
    var fune = new Fune();
    player1.addFune(fune);
}
```

クラスを応用すると、このように簡単に船を4つ作ることができます。

これだけではまだ使いたい船の切り替えはできません。操作をシンプルにしたいので、操作している船以外の船をタッチをしたらその船を操作できるようにします。

```
var Fune = Class.create(Sprite, {
中略                                    ◀ タッチイベントが起きたときのイベントハンドラを置く
    ontouchend: function(params) {
        if (this.player.getActiveFune() == this) {
                        ◀ 同じ船を触ったのでなにもしません
        } else {
            this.player.setActiveFune(this);
        }
    },                  違う船に切り替えます
});
```

実際はGameManagerとPlayerなどにほかの細かい変更は色々あるのですが、詳細は「chapter_10 → 12」で確認してみてください。

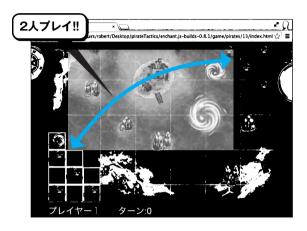
2人目のプレイヤーを作る

次は2人目のプレイヤーを実装します。これができると対戦型ゲームの一方手前まで近づきます。船をクラスで追加したように、プレイヤーもクラスにしてあるので、増やすのは簡単です。

```
// プレイヤー1
var player1 = new GamePlayer({name:"プレイヤー1"});
manager.addPlayer(player1);

// プレイヤー1に船を4つあげよう  ◀ プレイヤー1に船を4つ追加
for (var i=0; i <4; ++i) {
    var fune = new Fune();
    player1.addFune(fune);
}

// プレイヤー2
var player2 = new GamePlayer({name:"プレイヤー2"});
```

```
manager.addPlayer(player2);

// プレイヤー 2に船を4つあげよう
for (var i=0; i <4; ++i) {
    var fune = new Fune();
    player2.addFune(fune);
}
```

プレイヤー 2に船を4つ追加

2人プレイ!!

画面の下のほうに見えると思いますが、現在のターンのプレイヤー名を表示するために**TurnUI**と**GameManager**にちょっと手を入れました（chapter_10 → 13 のサンプル）。遊んでみれば、実際にそれぞれのプレイヤーのターンが変わります。あとちょっとで対戦できるようになりますね。

Chapter 11 ユニットパラメータ

本章で勉強すること

- ゲーム的なプログラムと玩具的なプログラムの違いは「自由」さと「制約」の違い
- 制約（＝ルール）を加えてゲーム度を高めよう
- 各ユニットにパラメータを用意する
- パラメータにはメソッドでアクセスするように指定しておくとあとで便利
- クラスを利用して、楽にユニットのバリエーションが作れるようにしよう

ここまでの開発では、ゲームの描画や操作に注力してきました。enchant.jsでのHelloWorldを表示したところから見るとかなり進んだと思います。でも、今の私たちのゲームの状態ではまだ「本当」のゲームとはいえません。それよりはどちらかというと「玩具」に近いものといえるでしょう。

玩具にはルールがありません。「自由」に楽しめるものです。一方、ゲームにはルールという「制約」

があって、その目的を達成することを楽しむものです。そこが玩具とゲームの根本的な違いといってよいでしょう。ということで、現在のゲームにさらにルールを実装していくことにしましょう。

ゲームのルール

ルールはいろいろと考えられますが､最初はできるだけシンプルなところから始めるのがよいでしょう。いろいろ進めていくうちに、徐々に拡張は可能です。

今のゲームにもすでに実装したルールがあります、それは「プレイヤーのターンは交互に換わる」というルールです。

たとえば､「コインで次のターンが決まる」や「ユニットの素早さでターンが決まる」などの可能性も考えられますが、今はシンプルに交互に進行していきます。このルールを壊さずに、今からユニットにパラメータを追加して、バトル、船の沈没、そして勝利条件を作り込んでいきます。

ユニットのパラメータ

シミュレーションゲームはほかのジャンルのゲームと比べて､パラメータ（各種の能力や状態を示す値）が多いのが特徴です。このゲームのパラメータも多いほうですが､最低限のもので済ませておきましょう。今回は次のようなパラメータを用意することにします。

- **移動力（movement）**
- **攻撃距離（range）**
- **攻撃力（attack）**
- **防御力（defense）**
- **体力（hp）**

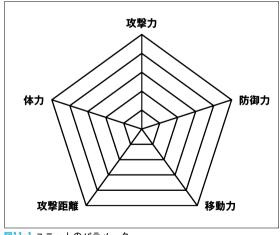



図11-1 ユニットのパラメータ

ユニットのコンストラクタの中で、これらのパラメータの定義と初期化を行いましょう。

```
var Fune = Class.create(Group, {
    initialize: function(id, stats) {
        中略
        this.stats = {
            movement:  3,
            range:     3,
            attack:  100,
            defense:  50,
            hpMax:   100,
        }
    },
```

移動力
攻撃距離
攻撃力
防御力
体力

このように書いておけば、次のように直接パラメータにアクセスできます。

```
var attack = fune.stats.attack;
```

しかし、あとで何か仕様変更があったら困るので、パラメータに直接アクセスするよりはメソッドでアクセスするように書いておきます。

```
var attack = fine.getAttack();
```

ゲームを作るときは仕様変更が付き物です。メソッドで書いておけば、パラメータの値をそのまま渡す以外にも、なにか加工をして渡すということもできるでしょう。それぞれのパラメータにアクセスするためにメソッドを定義します。

```
getMovement: function() {
    return this.stats.movement;
},
getRange: function() {
    return this.stats.range;
},
getAttack: function() {
    return this.stats.attack;
```

移動力を返すメソッド
攻撃距離を返すメソッド
攻撃力を返すメソッド

```
},
getDefense: function() {
    return this.stats.defense;
},
getHPMax: function() {
    return this.stats.hpMax;
},
getHP: function() {
    return this.stats.hp;
},
```

防御力を返すメソッド

体力（最大値）を返すメソッド

現在の体力を返すメソッド

パラメータの表示

ユニットにパラメータを実装しましたが、今のところプレイヤーがそれを確認する方法がありません。そこで、パラメータを確認できるように選択した船をタッチをするとステータスウィンドウが開くようにしてみましょう。

まずは、空のウィンドウを表示する機能と、それを閉じるためのボタンを実装します。

```
initialize: function(fune) {
    Scene.call(this);
    game.pushScene(this);

    var windowGroup = new Group();
    windowGroup.x = (960 -512)/2;
    windowGroup.y = (640 -512)/2;
    this.addChild(windowGroup);

    var windowSprite = new Sprite(512, 512);
    windowSprite.image = game.assets[uiWindowSprite];
    windowGroup.addChild(windowSprite);

    var self = this;
    var cancelBtnSprite = new Sprite(128, 64);
    cancelBtnSprite.image = game.assets[uiCancelBtnSprite];
    cancelBtnSprite.x = 64;
    cancelBtnSprite.y = 512 -96;

    windowGroup.addChild(cancelBtnSprite);

    cancelBtnSprite.addEventListener(enchant.Event.TOUCH_END,
    function(params) {
        game.popScene();
        if (self.onCancel) {
            self.onCancel()
        }
    });
},
```

512×512pxlのウィンドウをx=224, y=64の位置に配置正方形のウィンドウが中央に表示されます

キャンセルボタンの配置

ボタンが押されたときのイベントハンドラを定義

サンプルを実行してみると、これでウィンドウが開閉できるようになっているはずです。

次にウィンドウ上にパラメータを表示してみましょう。数字だけでは寂しいのでキャラクターも表示しましょう（サンプルは、chapter_11 → 14にあります）。

前略

```
var windowSprite = new Sprite(512, 512);
windowSprite.image = game.assets[uiWindowSprite];
windowGroup.addChild(windowSprite);

var statsGroup = new Group();
statsGroup.x = 64;
statsGroup.y = 32;
windowGroup.addChild(statsGroup);

var fontColor = "rgba(255, 255, 105, 1.0)";

captainLabel = new Label("船長："+fune.getCaptainName());
statsGroup.addChild(captainLabel);
captainLabel.x = 0;
captainLabel.y = 0;
captainLabel.font = fontStyle;
captainLabel.color = fontColor;

attackLabel = new Label("攻撃力："+fune.getAttack());
statsGroup.addChild(attackLabel);
attackLabel.x = 0;
attackLabel.y = 64 *1;
attackLabel.font = fontStyle;
```

船長の名前を表示

攻撃力

```
attackLabel.color = fontColor;

中略  ほかのパラメータも同じように実装していきます

hpLabel = new Label("HP："+fune.getHP()+"/"+fune.getHPMax());
statsGroup.addChild(hpLabel);
hpLabel.x = 0;
hpLabel.y = 64 *5;
hpLabel.font = fontStyle;
hpLabel.color = fontColor;

var pirate = new Sprite(400, 640);
pirate.opacity = 0;
pirate.x = 400;
pirate.y = -50;
pirate.image = game.assets[pirateSprites[0]];
windowGroup.addChild(pirate);
省略
```

体力の表示

キャラクターの表示

さらに、GameMapをタッチするとウィンドウを閉じるようにします。

```
ontouchend: function(params) {
    if (this.player.isActive()) {
        if (this.player.getActiveFune() == this) {
            var popup = new StatusWindow(this);
            popup.onCancel = function() {
                //ウィンドウは閉じました
            }
        } else {
            this.player.setActiveFune(this);
        }
    }
}
```

ユニットの種類を増やそう

これまでの実装で、4つの船を用意していました。でも、どれも同じパラメータを持っていました。本来は船の種類は4種類で4つのユニットあるはずです。そこでそれぞれのユニットを次のように実装します。

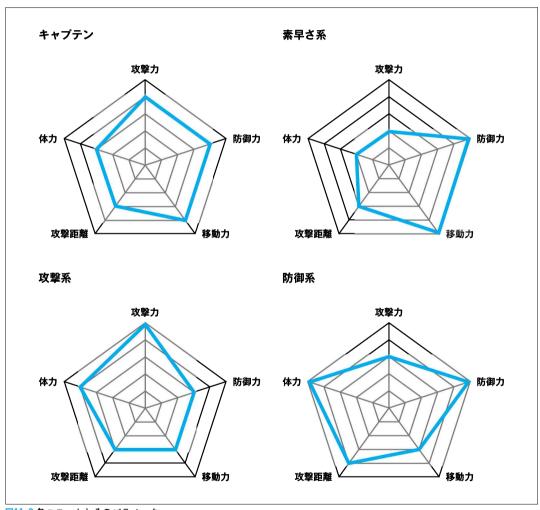



図11-2 各ユニットとそのパラメータ

どれも基本的な部分は同じですが、ちょっとずつ特徴が異なります。こういう場合、オブジェクト指向ではベース（基本）クラスを作り、それぞれのユニットクラスはベースを継承して実装します。

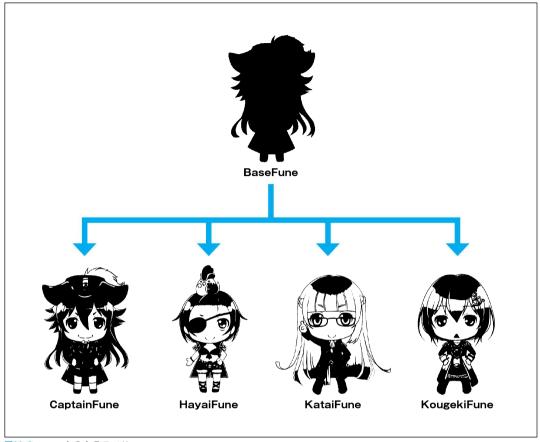



図11-3 ユニットのクラスツリー

まず、今の船のクラスを汎用的な名前「BaseFune」に変えます（サンプルはchapter_11 → 15にあります）。

```
var BaseFune = Class.create(Group, {
    initialize: function(id, stats) {
        Group.call(this);
```

それぞれの船はこの基本クラスを継承して作ります。実際はどの船も同じような作り方になりますので、ここでは1つだけ、キャプテンの船の作り方を見てみましょう。変更を加えるのはおもにパラメータの部分です。

```
var CaptainFune = Class.create(BaseFune, {
    initialize: function(id) {
        BaseFune.call(this, id, {
            movement:   3,
            range:      3,
            attack:   100,
            defense:   50,
            hpMax:    100,
        });

        this.fune.frame = [0, 0, 1, 1, 2, 1, 0, 0, 3, 3];
    },

    getCaptainName: function() {
        return "キャプテン";
    }
});
```

BaseFuneを継承してCaptainFuneを作る

パラメータを設定

名前を返すメソッド

あとは残りの3つの船を同じように作りこんでいきます。ゲームの初期化時の船の割り当てを変更して、それぞれのプレイヤーに4種類の船を持たせます。

Chapter 12

対戦ロジックを作る

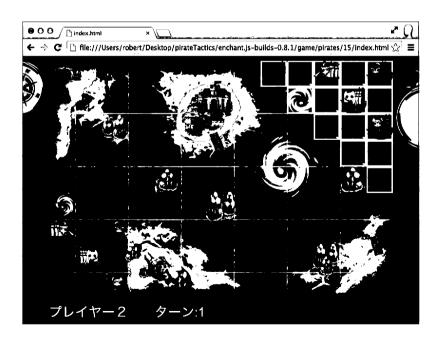
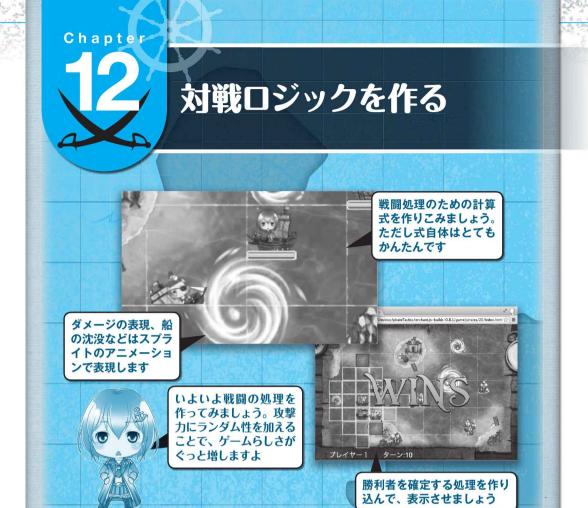
戦闘処理のための計算式を作りこみましょう。ただし式自体はとてもかんたんです

ダメージの表現、船の沈没などはスプライトのアニメーションで表現します

いよいよ戦闘の処理を作ってみましょう。攻撃力にランダム性を加えることで、ゲームらしさがぐっと増しますよ

WINS

プレイヤー1　ターン:10

勝利者を確定する処理を作り込んで、表示させましょう

本章で勉強すること

- ユニット同士の対戦時のルールを考えてみよう
- クリティカルヒット、ミスなどのランダムな要素を入れるとゲームが面白くなります
- 爆発や船が沈むなどのアニメーションを加えよう
- ユニットが全滅したら終了するようにして、ゲームを完結させよう

戦闘解決のルール

ユニットにパラメータが付きましたので、いよいよ対戦用の機能の実装に入れるようになりました。攻撃ルールの再確認と細かい部分の仕様を決めてから、実装に入ります。まずはルールを確認しましょう。

ルール1：ダメージ計算

ダメージの計算式は次のようなシンプルなものになります

自分の攻撃力 - 相手の防御力 = ダメージ

算出されたダメージはそのまま相手のHPから引き算をします。相手のHPが0になったら船は沈没です。

ルール2：ダメージにランダム性

シミュレーションゲームですから、ダメージには偶然によるちょっとしたにばらつきを付けたいと思います。どれくらいのばらつきをつけるかはゲームバランスのキモになりますが、一般的にばらつきの幅が広いと戦略が立ちにくくなります。シミュレーションゲームの場合はおよそ5 ～ 20%のばらつきを採用する場合が多いと思います。今回はとりあえず無難な10%のばらつきで行きましょう。

攻撃力には10%のばらつきがある

いまは10%にしますが、今後自分でいろいろなバランスを試してみて、体感的な影響を確認してみるとよいでしょう。

ルール3：クリティカルヒット

シミュレーションやRPGの定番ともいえる、運が良いときのクリティカルヒットも欲しいですね。
クリティカルヒットの効果はゲームによってさまざまですが、今回は、運が良いときに相手に与えるダメージが2倍になるというよくあるパターンでいきましょう。どれくらいの確率で発動するのが適切かは分かりませんが、今回は、

10%の確率でダメージが2倍

というクリティカルヒットが発生するようにプログラムします。

ルール4：攻撃が失敗することもある

攻撃が確実に当たると緊張感が削がれますので、たまに攻撃ミス（失敗）が起きるようにします。ミスもクリティカルヒットと同じく10%でいきましょう。

10%の確率で攻撃ミス

これで基本的な戦闘解決のルールは決まりました。それではプログラミングに入りましょう。

バトルルールの実装

バトルの実装は基本的に2つに分けます、攻撃のダメージを計算するメソッド（**attackFune**）とダメージを受けるメソッド（**takeDamage**）です（サンプルはchapter_12 → 16にあります）。

最初はさきほどのルールを実装した**attackFune**から作り始めます。この関数が基本的にダメージを計算するメソッドになります。

```
attackFune: function(otherFune) {
    var damage;
    var baseDamage = this.getAttack();
    var variance   = Math.random() -0.5;
    var variableDamage = (baseDamage /10) * variance;

    var attackRoll = Math.random();

    // クリティカルヒット 10%
    // ミス 10%
    if (attackRoll > 0.9) {
        // クリティカル ダメージx2
        damage = (baseDamage +variableDamage) *2;
    } else if (attackRoll < 0.1) {
        // ミス ダメージ 0
        damage = 0;
    } else {
        damage = baseDamage +variableDamage;
    }

    damage = Math.ceil(damage)
    var beforeHp = otherFune.getHP();
    var self = this;
    otherFune.takeDamage(damage, function(afterHp) {
        中略
    })
},
```

- 攻撃力から基本ダメージを算出
- Math.randomは0～1の範囲で乱数を発生させる -0.5することで-0.5～0.5の範囲になる
- 基本ダメージの1/10に対して乱数を適用し補正値を出す
- attackRollが0.9より大きいならダメージ2倍
- attackRollが0.1未満ならミス
- それ以外は基本ダメージ＋補正値で算出
- 小数部分を切捨て
- takeDamegeメソッド呼出し
- ダメージを受けたあとのコールバック

次は**attackFune**で計算されたダメージを受ける**takeDamage**のメソッドを実装します。

```
takeDamage: function(damage, onEnd) {
    var actualDamage = Math.max(damage -this.getDefense(), 1);
    this.stats.hp -= actualDamage;
    var self = this;

    onEnd();
},
```

ダメージ値から防御力を引く。1未満のときは1とする

体力からダメージを引く

処理が終ったのでコールバックを呼ぶ

今のところは非常にシンプルですが、このあとだんだん処理が増えてきます。

攻撃の距離を確認する

それぞれの船に攻撃距離のパラメータがありますので、それに従って攻撃したい相手との距離を計算し、攻撃できるかどうかを確認してから処理に入ります。withinRangeのメソッドをベースクラスに追加し、マップのタッチイベントで確認してからattackFuneを呼びます。withinRangeは次のような実装になります。

```
withinRange: function(i, j) {
    var distance = utils.getManhattanDistance(this.i, this.j, i, j);
    console.log("withinRange", "distance", distance, "range",
    this.stats.range, distance <= this.stats.range);
    if (distance <= this.stats.range) {
        return true;
    } else {
        return false;
    }
},
```

攻撃範囲内ならtrueを返す

攻撃範囲外ならfalseを返す

そしてGameMapのontouchendの中に攻撃の操作を組み込みます。

```
ontouchend: function(params) {
    if (this.player.isActive()) {
```

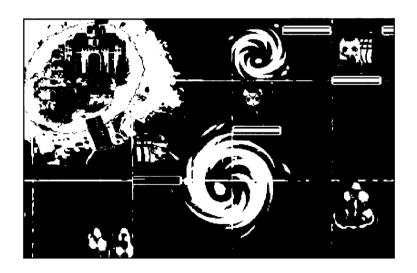
中略

```
        } else {
            var activePlayer = this.player.controller.getActivePlayer();
            var activeFune   = activePlayer.getActiveFune();
            if (activeFune.withinRange(this.i, this.j)) {
                activeFune.attackFune(this);
            }
        }
    },
```

withinRangeの結果がtrueだったら

処理を実行する

HPゲージを作りましょう

攻撃のあとは残りHPを簡単に確認したいのでHPのゲージを作ります（サンプルはchapter_12 → 17にあります）。

このようなゲージは実際３つの画像から構成しています。バックの画像、HPの残りを示す緑、ダメージをあらわす赤です。状況に応じてこの画像を変形してゲージを表現します。

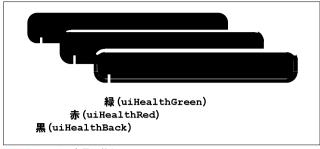



図12-1 ゲージの表示の仕組み

最初は簡単にこの3つの画像でBaseFuneにHPのゲージを作りましょう

```
var BaseFune = Class.create(Group, {
    initialize: function(id, stats) {
        中略
        var healthBackSprite   = new Sprite(64, 12);
        this.healthBackSprite  = healthBackSprite;
        healthBackSprite.image = game.assets[uiHealthBack];
        healthBackSprite.x     = 16;
        healthBackSprite.y     = 96 -12;
        this.addChild(healthBackSprite);

        var healthRedSprite   = new Sprite(64, 12);
        this.healthRedSprite  = healthRedSprite;
        healthRedSprite.originX = 0
        healthRedSprite.image = game.assets[uiHealthRed];
        healthRedSprite.x     = 16;
        healthRedSprite.y     = 96 -12;
        this.addChild(healthRedSprite);

        var healthGreenSprite   = new Sprite(64, 12);
        this.healthGreenSprite  = healthGreenSprite;
        healthGreenSprite.originX = 0
        healthGreenSprite.image = game.assets[uiHealthGreen];
        healthGreenSprite.x     = 16;
        healthGreenSprite.y     = 96 -12;
        this.addChild(healthGreenSprite);
        略
```

黒の読み込み

赤の読み込み

緑の読み込み

さきほど作ったtakeDamageのメソッド内でゲージの状態を計算して更新するようにします。updateHPBarを呼び出します。

```
takeDamage: function(damage, onEnd) {
    var actualDamage = Math.max(damage -this.getDefense(), 1);
    this.stats.hp -= actualDamage;
    var self = this;
    this.updateHPBar(function() {   ◀ゲージを更新するメソッドupdateHPBarを実行
        onEnd();
    });
},
```

updateHPBarは次のようになっています。

```
                                        ダメージの割合を計算
updateHPBar: function(onEnd) {
    var ratio = Math.max(this.getHP() / this.getHPMax(), 0);
    if (ratio > 0.5) {
        this.healthGreenSprite.scaleX = ratio;   ◀割合に合わせてゲージを縮小
    } else {
        this.healthGreenSprite.scaleX = 0;   ◀ヒットポイントが半分を切っていれば
                                               緑のゲージは表示させない
    }
    this.healthRedSprite.scaleX = ratio;   ◀赤いゲージを縮小する
    onEnd();
},
```

攻撃のエフェクト

ロジックの実装さえできればいちおう遊べますが、それだけだとプレイヤーに分かりにくいので攻撃が当たるとダメージを数字化した表現を画面に出すようにします。ちょっとした爆発のアニメーションを再生しましょう（サンプルはchapter_12 → 18にあります）。

Spriteを継承した爆発のクラスを定義します。

```
var Explosion = Class.create(Sprite, {  ◀爆発クラス
    initialize: function(id, stats) {
        Group.call(this, 32, 32);
        this.image = game.assets[shipsSpriteSheet];
        this.frames = [0,1,2,3,0,1,2,3,0,1,2,3,4,5,null];
        this.counter = 0;
    },
    onenterframe:function(){  ◀最後のフレームになったらスプライトを消す
        this.counter++;
        if (this.counter == this.frames.length -1 ) {
            this.parentNode.removeChild(this);
        }
    },
}
```

takeDamageのあとに**explosion**を実行します。アニメーションが終ったら自分で**removeChild**をしてくれるので、終了のための処理は書かなくても大丈夫です。

```
attackFune: function(otherFune) {
    中略
    otherFune.takeDamage(damage, function() {
        var explosion = new Explosion();  ◀爆発を呼び出す
        game.currentScene.addChild(explosion);

        explosion.x = otherFune.x;
        explosion.y = otherFune.y;
    })
},
```

船の沈没

ゲージはできましたが、現在の状態ではある船のHPがゼロ以下になっても終わらず、そのまま戦い続けます。そこで、HPが0になったときに船が沈没するようにしましょう。

これは爆発と同じようにアニメーションを一度だけ実行します。フレームごとにアニメーションを進めて行きます。

```
sinkShip: function(onEnd) {
    this.frames = [4,4,4,5,5,6,6,7,7,8,8];  ◀ 水没アニメーションを表示します
    this.counter = 1;
    this.onenterframe = function(){ // enterframe event listener
        this.counter++;
        if (this.counter == this.frames.length ) {
            this.player.removeFune(this);  ◀ アニメーションが完了したら、船はゲームから
                                              外して次のプレイヤーにターンを回します

            this.parentNode.removeChild(this);
            onEnd();
        }
    },
}
```

勝利条件

現在はどちらかの船が全滅するまでは遊べますが、実際の勝利条件はプログラムしていないので、その後ゲームは進行不能の状態になります。それをこれから直しましょう（サンプルはchapter_12 →20にあります）。

基本的にシミュレーションゲームには勝利条件があります。

たまにミッションが終了条件なしに続くケースもありますが、ほとんどは相手を全滅するか、何かの目標を達成すれば終了します。今回のゲームはシンプルにしておきたいので勝利条件を全滅だけにします。いまのゲームの作りだと勝利条件を一番適切に確認できるのは`nextTurn`の中です。

```
endTurn: function() {
    var player = this.getActivePlayer();
    player.setActive(false);

    var winner = this.getWinner();    勝利者がいるか調べる
    if (winner) {    いた場合
        var playerBanner = new Sprite(512, 256);    勝利者表示用の
                                                    スプライトを作る
                                                    プレイヤー 1をセット
        if (player.id == 1) {
            playerBanner.image = game.assets[uiPlayerBanner1];
        } else if (player.id == 2) {
            playerBanner.image = game.assets[uiPlayerBanner2];
        }
                                                    プレイヤー 2をセット

        playerBanner.opacity = 0;
        playerBanner.x = 480 -256;    プレイヤー名を表示
        playerBanner.y = 320 -128;
        game.currentScene.addChild(playerBanner);

        var resultBanner = new Sprite(512, 256);    Winを表示
        resultBanner.image = game.assets[uiWin];
        resultBanner.opacity = 0;
        resultBanner.x = 480 -256;
        resultBanner.y = 320 -128;

        game.currentScene.addChild(resultBanner);
    } else {    勝利者がいなければ
        this.turnCounter++;
        var playerBanner = new Sprite(512, 256);

        if (player.id == 1) {
            playerBanner.image = game.assets[uiPlayerBanner2];
        } else if (player.id == 2) {
            playerBanner.image = game.assets[uiPlayerBanner1];
        }
                                                    プレイヤー名を表示して

        playerBanner.opacity = 0;
        playerBanner.x = 480 -256;
        playerBanner.y = 320 -128;
        game.currentScene.addChild(playerBanner);

        this.startTurn();    次のターンへ
```

```
            utils.endUIShield();
            game.currentScene.removeChild(playerBanner);
        }
    },
```

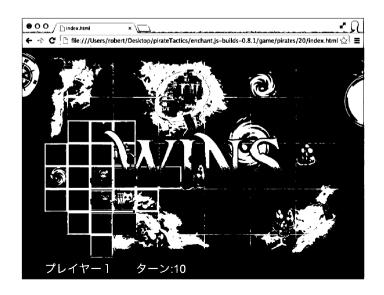

getWinnerではどちらかのプレイヤーの船が0になったかどうかを調べ、条件を満たしたら勝利者を返します。

```
    getWinner: function() {
        if (this.getActivePlayer().getFuneCount() == 0) {
            if (this.getNonActivePlayer().getFuneCount() == 0) {
                return this.getActivePlayer();
            } else {
                return this.getNonActivePlayer();
            }
        } else if (this.getNonActivePlayer().getFuneCount() == 0) {
            return this.getActivePlayer();
        }
        return null
    },
```

現在のプレイヤーの船が0で相手も0なら自分の勝利

そうでなければ相手の勝利

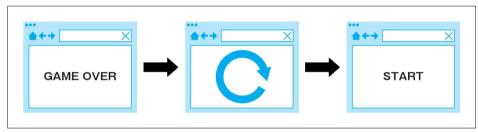
相手の船が0なら現在のプレイヤーの勝利

勝利者はいません

ゲームを再開

勝利者が確定するとゲームが進行不能になります。問題は終了後何もすることがないことです。本当はいろいろな画面とロジックでゲームを初期状態に戻して再開するとよいのですが、実装すると複雑ではないわりにたくさんのコードを説明しなければなりません。

ゲームのキャンペーンモードについては今後パート4で対応する予定ですので、この時点ではブラウザ機能を活かして、終了後、画面をタッチしたらWebページをリロードするようにしてみましょう。

図12-2 ゲームの終了と再読み込み

```
endTurn: function() {
    var player = this.getActivePlayer();
    player.setActive(false);

    var winner = this.getWinner();
    if (winner) {   ◀ 勝利者がいれば
        中略

        location.reload();   ◀ これはブラウザの特別なリロード機能です。
                               enchant.jsの関数ではありません
    } else {
        中略
    }
},
```

ここまででゲームが終了し、再開するまでを一通り作ることができました！

でも、見栄えの良いゲームにするにはまだまだやることあります。本書のパート3では、一度完成したゲームにさまざまなエフェクトを加えるなどの改良を行うことにします。

Part

3 エフェクトを追加しよう

本章で勉強すること

- シンプルなゲームでも作り込みが良いと、爽快感アップ
- enchant.jsのTimelineアニメーション機能を活用しよう
- イージング（easing）を使って動きに変化をもたせます
- スプライトの重なり方を正しくする方法を考えよう

　ゲームを作っていると「爽快感が大切」とよく聞きます。ゲームを進めるのが心地良いと、ユーザーが積極的にどんどんゲームを進めてくれます。でも、爽快感とはどんなものなのでしょうか？ ときどき「爽快感は派手さだ！」と言う人もいますが、ただの派手さだけでは爽快感に繋がらないはずです。

　ゲームの爽快感とは「操作」に対する「反応」の面白さとも言えるのではないでしょうか。操作に対する反応が適切であれば、ゲームはさらに楽しく、分かり易くなります。

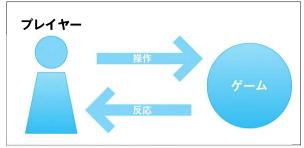



図13-1 アクションとリアクション

爽快感と磨き

爽快感のあるゲームは「磨かれている」ともいいます。磨きはゲームにとってはプラスα的なものでしかないかもしれません。しかし、たとえどんなにシンプルなゲームでもしっかりと磨かれていると、それなりにちゃんとしたゲームに見えます。逆にいくらよく考えられたゲームでも、磨きが足りないと地味に見え、ほとんどの人からはあまり評価されません。

enchant.jsのTimelineアニメーション

ゲームの爽快感を上げる手段はたくさんありますが。今回は誰でもできる簡単な手段でゲームの爽快感を改善したいので、enchant.jsのTimelineアニメーションという機能を紹介したいと思います。Timelineはあまりくどくどと説明をするよりはさっそく使ってみたほうが理解しやすいでしょう。たとえば、あるスプライトを30フレームのあいだにx軸方向にプラス100動かしたいとき、

```
mySprite.tl.moveBy(100, 0, 30);
```

と記述します。こうすれば毎フレームにつき`mySprite`はx軸方向に3.333ずつ動きます（100/30frame = 3.333...）。

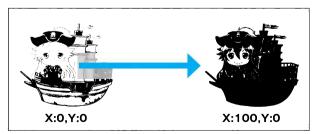



図13-2 TimelineによるX軸方向への動き

船を図のようにY方向にも動かしたければ、moveByを連続して使います。

```
mySprite.tl.moveBy(100, 0, 30).moveBy(0, 100, 30);
```

こうすればXを100移動したあとにYを100移動します。

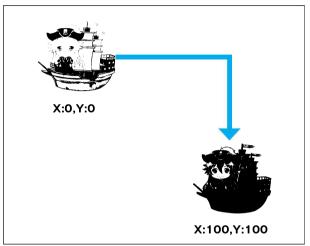



図13-3 X軸／Y軸の動きの組み合わせ

アニメーションとアニメーションのあいだにちょっとした停止期間を作りたいときはdelay()を使えます。たとえば、X軸方向の移動のあと10フレームのあいだ動きを止めるためには、

```
mySprite.tl.moveBy(100, 0, 30).delay(10).moveBy(0, 100, 30);
```

と記述します。アニメーションを同時に2つ連携して動かしたいときは .and() を使います。

```
mySprite.tl.moveBy(100, 0, 30).and().rotateBy(90, 30);
```

こうすることで、X軸方向とY軸方向同時に、ユニットが斜めに動いていきます。

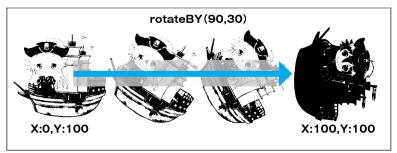



図13-4 船アイコンの回転

Timelineのもうひとつの重要な機能は、移動が終ったところでなにか別の処理を実行できる`.then()`です。これを使うと移動後に自由な処理を実行させられます。

```
mySprite.tl.moveBy(100, 0, 30).then(function() {
    scene.removeChild(mySprite);
})
```

移動を終ったらスプライトを削除します

最後に説明したいTimelineの機能は**イージング**（Easing）です。ここまでの変更は毎フレーム同じ量でしたが、イージングでだんだん早くなる、またはだんだん遅くなるなどのアニメーションが作れます。

```
this.tl.moveTo(100, 0, 30, enchant.Easing.QUAD_EASEIN)
```

上記の場合Xの移動スピードは直線ではなく、だんだんと早くなります。

動きの種類は`QUAD`（緩やかになる）、`BOUNCE`（跳ねる）、`ELASTIC`（行き過ぎて戻る）があり、「_」（アンダースコア）でつないで、`EASEIN`（動きの始め）、`EASEOUT`（動きの終わり）、`EASEINOUT`（両方）を指定する方式になっています。

▼ 表13-1 イージングの設定

	easeIn	easeOut	easeInOut
Quad	easeInQuad	easeOutQuad	easeInOutQuad
Bounce	easeInBounce	easeOutBounce	easeInOutBounce
Elastic	easeInElastic	easeOutElastic	easeInOutElastic

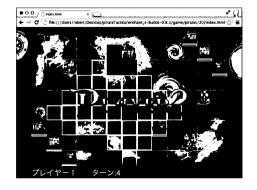
enchant.jsでイージングに設定できる式はたくさんありますが、一気に覚える必要はありません。使っているうちに覚えていけばよいでしょう。また、ほとんどのフレームワークやエンジンでは同じようなイージングのカーブが指定できます。

ここまで見たTimelineの機能を使ってゲームの爽快感を改善していきましょう。

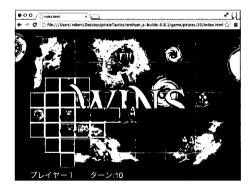
UIの改善

最初は簡単なところから始めたいのでUIの改善をしましょう。ターンが変わるときのメッセージをフェードインするようにします（サンプルはchapter_13 → 20にあります）。**tl**に対して**fadeIn**（フレーム数）といった形で指定します。

```
endTurn: function() {
    var player = this.getActivePlayer();
    player.setActive(false);

    var winner = this.getWinner();  ◀ 勝者を取得
    if (winner) {
        中略
    } else {  ◀ 勝者がまだいなければ
        中略

        var self = this;  メッセージのフェードインとアウトを指定
        playerBanner.tl.fadeIn(10).delay(15).fadeOut(5).then(function() {
            self.startTurn();
            utils.endUIShield();  ◀ モーダルシールドオフ
            game.currentScene.removeChild(playerBanner);
        })
    }
},
```

サンプルでは**utils.beginUIShield**と**utils.endUIShield**というメソッドが新たに作られています。**beginUIShield**を実行すると、目に見えない（**opacity = 0**の）スプライトがゲーム画面全体を覆って、プレイヤーの操作を遮断します。これはモーダルシールドという手法です。モーダル（modal）は「モード（mode）がある」と言った意味で、モードが切り替わると適切な応答が返るまでプレイヤーに操作をさせないようにしています。**endUIShield**メソッドを実行するとシールドがオフになります。

勝利メッセージもフェードイン／アウトでポップアップするようにします。

```
endTurn: function() {
    var player = this.getActivePlayer();
    player.setActive(false);

    var winner = this.getWinner();
    if (winner) {  ← 勝者が確定していれば
        中略
        var self = this;  プレイヤー名をフェードイン／アウト
        playerBanner.tl.fadeIn(10).delay(15).fadeOut(5).then(function() {
            game.currentScene.removeChild(playerBanner);

            var resultBanner = new Sprite(512, 256);
            resultBanner.image = game.assets[uiWin];
            resultBanner.opacity = 0;
            resultBanner.x = 480 -256;
            resultBanner.y = 320 -128;
            game.currentScene.addChild(resultBanner);
            ← 「Wins」と表示するメッセージを作り

            resultBanner.tl.fadeIn(10).delay(45).fadeOut(5).then(function() {
                location.reload();  フェードイン／アウトさせます
            })
        });
```

13

ステータスウィンドウの表示にもイージングを追加しましょう。ステータスウィンドウを開くときはだんだんと現れるようにして、ELASTIC_EASEOUTを指示します（サンプルはchapter_13 → 21にあります）。

```
var FunePopup = Class.create(Scene, {
    windowGroup.tl.scaleTo(1, 10, enchant.Easing.ELASTIC_EASEOUT).
    then(function() {
        中略
    })
},
```

ウィンドウの「戻る」ボタンに触ったときの反応も付けましょう。

```
windowGroup.tl.scaleTo(1, 10, enchant.Easing.ELASTIC_EASEOUT).
then(function() {
    中略
    cancelBtnSprite.addEventListener(enchant.Event.TOUCH_START,
    function(params) {
        cancelBtnSprite.tl.scaleTo(1.1, 10,
        enchant.Easing.ELASTIC_EASEOUT)
    });

    cancelBtnSprite.addEventListener(enchant.Event.TOUCH_END,
    function(params) {
        shieldSprite.tl.fadeTo(0, 5);
        cancelBtnSprite.tl.scaleTo(0.9, 3).and().fadeTo(0, 5);
        pirate.tl.fadeTo(0, 5);
        windowSprite.tl.fadeTo(0, 5).then(function() {
            game.popScene();
            if (self.onCancel) {
                self.onCancel()
            }
        });
    });
})
```

マウスボタンを押したときにボタンがすこし大きくなる

離すとウィンドウが消えるようにする

船の移動を改善

船が瞬間移動をしているところをスイスイ移動するアニメーションに変えると見栄えがぐっと良くなります（サンプルはchapter_13 → 22にあります）。

```
moveFune: function(fune, i, j, onEnd) {
    var postion = this.getMapPositionAtTile(i, j);
    var worldPosition = this.toWorldSpace(postion.localX, postion.localY);

    worldPosition.x -= 16;
    worldPosition.y -= 32;

    距離によって移動時間が変わるようにします
    var distance = utils.getEuclideanDistance(fune.i, fune.j, i, j);

    fune.i = i;
    fune.j = j;

    距離によって移動時間を変えることで大体同じスピードになります
    fune.tl.moveTo(worldPosition.x, worldPosition.y, distance *10,
    enchant.Easing.QUAD_EASEINOUT).then(onEnd);
},
```

ただし、サンプルを触ってみると分かるのですが、船が移動しているあいだに違うエリアをタッチすると不具合が発生します。これを防ぐ方法はいろいろありますが、以前使ったモーダルシールドと同じ手法を使いたいと思います。

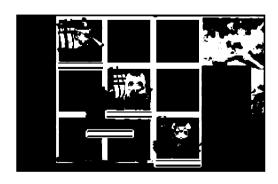
```
ontouchend:function(params) {
    中略
    if (this.getManhattanDistance(this.activeFune.i, this.activeFune.j,
    tile.i, tile.j) <= this.activeFune.getMovement()) {
        var self = this;
        utils.beginUIShield(); モーダルシールドをon
        self.moveFune(self.activeFune, tile.i, tile.j, function() {
            var specialTile = self.getSpecialTile(tile.i, tile.j);
            if (specialTile) {
                specialTile.use(self.activeFune, function(needsRemove)
{
                    if (needsRemove) {
                        self.removeSpecialTile(specialTile);
                    }
```

```
                        utils.endUIShield();  モーダルシールドをoff
                        self.controller.endTurn();
                    });
                } else {

                    utils.endUIShield();  モーダルシールドをoff
                    self.controller.endTurn();
                }
            });
        }
        (中略)
    },
```


シールドを使っておくと移動中に画面のほかのところに触れても不具合は発生しません。

また、サンプルコードではthenの中を変更しています。アニメーションが終って初めてターンが進むようにするため、ターンを進める処理をthenの中に移動しました。こうすればアニメーションが終るまで、ターンは進まないことになります。GameMapのmoveFuneにタイムラインを適用します。

ユニットの重なり方を改善する

これで船が滑らかに動くようになりました。しかし、サンプルで気が付いたかもしれませんが、ときどき船の移動中に変な重なり方になることがあります。移動中の船がほかの船の上に重なるか下に重なるかは船によってまちまちです。

これは画面の奥行きに関する描画順の問題です。この問題は多くの場合、z-sortという仕組みを作って解決します。

今まではX軸とY軸で船を操作してきました。奥行きについては通常Z軸で取り扱います。Z軸の位置関係を見て描画の順番を決めるのがz-sortですが、enchant.jsにはZ軸やz-sortの概念はありません。

ただし、幸いにも実装は簡単に直すことができます。今回のゲームの画面ではスプライトは左にあるものから順に奥に並んでいるということになります。つまり、Y軸の並び順を調べればZ軸の順番も決められることになります。z-sortはy-sortで置き換えられるようになりますので、y-sortで描画順を決めましょう（サンプルはchapter_13 → 23にあります）。

13

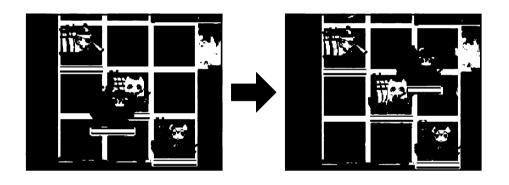
```
zsort: function() {
    var zorder = [];
    for (var c=0; c < this.playLayer.childNodes.length; ++c) {
        zorder.push(this.playLayer.childNodes[c]);
    }
    zorder.sort(function(a, b) {    ← JavaScriptの配列のソート機能を使います

        if (a.y > b.y) {
            return 1;
        } else if (a.y == b.y) {
            if (a.x > b.x) {
                return 1;
            } else if (a.x == b.x) {    ← ソートの順を決めます
                return 0;
            } else {
                return -1;
            }
        } else {
            return -1;
        }
    });

    for (var i=0; i < zorder.length; ++i) {    ← 順番が決まったら、それぞれを順に再登録していきます
        this.playLayer.addChild(zorder[i]);
    }
}
```

zorder.sort(function(a, b) {の部分で定義される関数で比較のルールを定義します。コードではaのY座標がbのY座標よりも大きければ1、同じ場合はX座標も比較して大きければ1、一致していれば0、

それ以外は-1が返るようになっています。1が返った要素は前に、0の場合はそのまま、-1の場合は後ろにします。sortメソッドはこのルールに従って配列の要素をすべて並べ替えてくれます。

そして、フレームごとにGameMapのzsortを呼ぶようにします。

```
tiles.addEventListener(enchant.Event.ENTER_FRAME, function(params){
    self.zsort();
})
```


これでちゃんと移動中の描画順が守られて不自然に見えません。

ほかにも細かいアニメーションを付けましたが、たとえばHPゲージや攻撃のエフェクトなども紹介した例と同じようになりますので、それぞれについてはサンプルコードで確認してみてください。

Point

このチャプターではゲームの見栄えだけを改善してルールのほうは一切変えていませんが、一気にゲームらしくなりました！
それは爽快感の効果です。

ゲームらしさがぐっと上がります！

本章で勉強すること

- サウンドを加えるとゲームの爽快感がぐっとアップ
- サウンドを管理するためのクラス（SoundManager）を作って一括管理しよう
- 音声の再生、一時停止、同時再生、ボリュームの調整方法を覚えよう
- インターネットのフリー素材などを活用して効果をアップしよう

見た目の磨きに続いてサウンドが加われば、ゲームの仕組みはそのままでも爽快感がもっと上がるはずです。ブラウザ内で再生するサウンドは、現在のところグラフィックの描画ほど高速な処理ができるわけはありませんが、ゲームに必要なちょっとした音声の再生であれば十分に利用可能です。

音声の再生の方法

まずは一番インパクトのある、BGM の再生から始めます。最初は画像と同じように音のデータをあらかじめ`preload`で読み込みます。

```
game.preload('../../../resources/sound/highseas.mp3');
```

こうしておけばグラフィックのときと同じように`game.onload`が呼ばれたあとは自由に再生ができます。

```
game.assets['../../../resources/sound/highseas.mp3'].play();
```

今回の音楽はちょっと贅沢にこのゲームのために作られました。しかし、自分で音楽は作れなくても、ゲームの音楽を作ってみたいという優秀なインディーズのクリエイターもいます。ゲームの企画が面白ければ協力者を見つけられるかもしれません（音楽に関してはTwitterなどのSNSで交流するのもお勧めです）。

SoundManagerを作る

ゲームに音声を付けるときの管理、とくに音量などは複雑になりがちですので、`SoundManager`を作って管理はそちらで統一することにしましょう。最初はたんにBGMを流すメソッドを用意します（サンプルはChapter_14 → 24にあります）。

```
var SoundManager = Class.create({
    initialize: function() {
        this.volume = 0.5;
        this.bgmPlaying = false;
    },

    playBGM: function() {
        this.bgmPlaying = true;

        game.assets[sndBGM].play();
        game.assets[sndBGM].volume = this.volume;
```

ボリュームの値の設定

再生開始

ボリュームの適用

```
        },
    })
```

SoundManagerのメソッドplayBGMを実行すればサウンドが鳴り出します。

サウンドを止める、一時停止させる

短い音の再生であれば、音の停止を気にする必要はほとんどないかもしれませんが、BGMのように場面に応じて音楽を切り替えるものは、明確に音を停止することが望ましいときもあるでしょう。今回のゲームではそこまで細かな制御が必要ではありませんが、念のためSoundManagerにstopとpauseを実装しておきます。

14

```
    var SoundManager = Class.create({
        中略
        pauseBGM: function() {  ◀ポーズ
            this.bgmPlaying = false;
            game.assets[sndBGM].pause();
        },

        stopBGM: function() {  ◀ストップ
            this.bgmPlaying = false;
            game.assets[sndBGM].stop();
        },
    })
```

名前のとおりstopはサウンドの再生を停止します、その後、playを実行すればまた最初から再生が始まります。一方、pauseは音を中断します。playを実行すれば中断した位置から再生を再開します。

効果音を実装

同じサウンドを複数回、同時に再生したいとき、たとえばゲームの中で撃つ弾の効果音などはサウンドを複製（clone()）してからplay()を実行します。

```
var sound = game.assets['se6.mp3'].clone();
sound.play();
```

このコードにおけるgame.assets['xxx.mp3']にはSoundオブジェクトが格納されており、clone()はそれを複製するためのメソッドです。soundオブジェクトを複製しなくても再生はできますが、その場合は同じ音声を重ねて鳴らすことはできません。サウンドを複製して再生する機能をSoundManagerに追加しましょう。


```
var SoundManager = Class.create({
    中略
    playFX: function(name) {
        var fx = game.assets[name].clone();   ← サウンドを複製
        fx.volume = this.volume;
        fx.play();
    },
})
```

これで爆発、水没、クリックなどのさまざまな効果音をSoundManagerのplayFXメソッドで再生できるようになります。

```
sndManager.playFX(sndExplosion);
```

ゲームにサウンドを付けるだけで倍ぐらい魅力がアップしますから、ゲームを作るときはぜひサウンドを付けるようにしましょう。

とくに効果音については、ネット上を探すことでフリー素材がいろいろ見つかります。もちろん良い効果音を選ぶのは簡単ではありませんが、ちょっとした手間と時間をかければ簡単にサウンドを追加でき、よりいっそう豪華になります。

音量調整

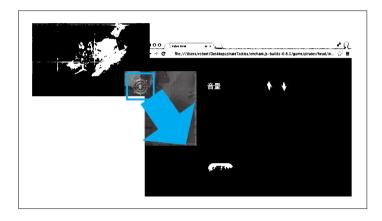
音量の調整はsoundオブジェクトのvolumeプロパティを用います。

volumeの値は0（0%）から1（100%）のあいだで指定します。初期値は1となっているのでさきほどはSoundManagerの音量を0.5に変えました。こうしておけばいきなり大音量で再生されることはないでしょう。

ここでは音量の調整用のメソッドを用意します。volumeUpとvolumeDownメソッドを用意し、volumeプロパティを5%ずつ変更します。値が変わったら確認用にクリック音を再生するようにします。

```
var SoundManager = Class.create({
    中略
    volumeUp: function() {
        this.volume += 0.05;  ◀ボリュームを5%アップ
        if (this.volume > 1) {
            this.volume = 1;
        }
        console.log("volume", this.volume);
        game.assets[sndBGM].volume = this.volume;  ◀適用
        this.playFX(sndClick);
    },

    volumeDown: function() {
        this.volume -= 0.05;  ◀ボリュームを5%ダウン
        if (this.volume < 0) {
            this.volume = 0;
        }
        console.log("volume", this.volume);
        game.assets[sndBGM].volume = this.volume;  ◀適用
        this.playFX(sndClick);
    },

    getVolume: function() {
        return this.volume;
    },
})
```

サウンドの設定画面

　これでプログラムからボリュームを調整する仕組みができました。次に音量はプレイヤー自身が調整できるように設定画面を作ることにしましょう。

```
var SettingsWindow = Class.create(Scene, {
    initialize: function(gameManager) {
        中略
        soundLabel = new Label("音量");
        settingsGroup.addChild(soundLabel);
        中略
        var sndUpButton = new Sprite(64, 64);
        settingsGroup.addChild(sndUpButton);
        中略

       sndUpButton.addEventListener(enchant.Event.TOUCH_END,
       function(params) {
           if (gameManager.sndManager.getVolume() < 1) {
               if (isKeyPressed == true) {
                   gameManager.sndManager.volumeUp();
                   sndUpButton.tl.scaleTo(1.0, 3).then(function() {
                   isKeyPressed = false;
                   });
               }
           }
       });
```

音量アップボタン

```
            var sndDownButton = new Sprite(64, 64);  ◀ 音量ダウンボタン
            settingsGroup.addChild(sndDownButton);
            中略
            sndDownButton.addEventListener(enchant.Event.TOUCH_END,
             function(params) {
                if (gameManager.sndManager.getVolume() > 0) {
                     if (isKeyPressed == true) {
                         gameManager.sndManager.volumeDown();
                         sndDownButton.tl.scaleTo(1.0, 3).then(function() {
                           isKeyPressed = false;
                         });
                     }
                }
            });
            中略
        },
    })
```


ループ再生

これでいったんサウンド再生機能の実装は完了したように見えるかもしれませんが、数分するとBGMは終ってしまいます。それを防ぐためにサウンドのループを考えましょう。

パート1でも述べたように、ブラウザの仕様は種類によっていろいろと異なります。これは音に関しても同じで、仕組みが大きく異なることがあります。基本的にブラウザはWebAudioSoundまたはDOMSoundという仕様のどちらかに従って実装されていることが多いのですが、基本のところはenchant.jsが吸収してくれます。

図14-1 サウンド再生機能とenchant.js

ただし、enchant.jsは音楽のループ再生には対応していません。そのため、音声再生の仕組みによってそれぞれ必要な対応が異なってきます。

DOMSound使用時は`enterframe`イベントが発生するたびに毎回`play`を呼び出すことでループを実現できます。しかし、WebAudioSoundでは`play`を呼ぶと先頭から再生されるため、このコードではループ再生になりません。WebAudioSoundの環境で前述の方法を実行するとノイズのような音になってしまい

ます。WebAudioSound使用時は`src.loop`プロパティに`true`を設定します。こうしておくと`play`を繰り返し呼び出す必要はありません。

両方の仕様に対応するにはどうすればよいでしょうか？ 今回の場合は、`src`プロパティがあるかないかを見て判別すればよいでしょう。次のコードでは、`if(game.assets[sndBGM].src)`という部分で、判断しています。`src`がない場合は`game.assets[sndBGM].src`のところが`false`になりますので、これで判断ができるというわけです（サンプルはchapter_14 → 25にあります）。

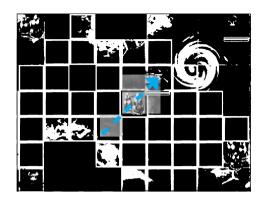
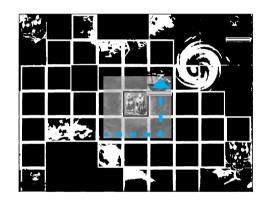
```
var SoundManager = Class.create(Sprite, {
    中略
    playBGM: function() {
        this.bgmPlaying = true;

        game.assets[sndBGM].play();
        if(game.assets[sndBGM].src){   ← srcプロパティの有無を調べる
            game.assets[sndBGM].src.loop = true;
        } else {
            game.currentScene.addChild(this);
        }
        game.assets[sndBGM].volume = this.volume;
    },
    中略
    onenterframe: function(){
        if (this.bgmPlaying) {
            game.assets[sndBGM].play();
        }
    },
```


スマートフォンでサウンドを使う

最後にスマートフォンでのサウンド再生に関してすこし補足しておきます。AndroidやiOSはまだHTML5の<audio>タグの実装が追いついていなかったり、特殊な仕様を採用していたりするという事情があります。そのため、enchant.jsの音声の再生はすべてのプラットフォーム／バージョンをサポートできているわけではありません。最新のモデルでなければ「動けばラッキー」くらいに思っておいたほうがよいでしょう。とくに、iOS上のブラウザ（Mobile Safari）については「プレイヤーが画面にタップしたとき以外、音声が再生できない」という大きな制限があります。

本章で勉強すること

- 移動力と経路を計算して正しく移動ができるようにしよう
- 経路の計算には定番のAスターサーチアルゴリズムを採用します
- でも、自分で実装する必要はなくフリーのライブラリを活用することにします
- 移動力に応じてユニットも動くようにし、動きにも変化を付けてみます

このゲームを作り始めて間もないころからフィールド上の陸と岩にユニットを置くことはできない仕様にしていました。しかし、気が付いた方もいると思いますが、移動の途中で岩や島を通過することは可能でした。

もちろん、これが飛行機なら問題ありませんが、やはり船は岩の周りを次のように回ってほしいものです。

そしてもちろん実際に移動できる距離も障害物に合わせて正しく計算する必要があります。今度はその部分の作り込みに取り掛かりましょう。

Aスターサーチ

地形の原点から目的地までの道を探すのは**探索**という操作になります。このような道を探索する**アルゴリズム**（計算式）はたくさんありますが、ゲーム業界で使われているアルゴリズムは基本的にひとつに絞られるようです。そのアルゴリズムは**Aスターサーチ**（A-star search）と呼ばれています。

Aスターがゲームで採用される理由はほかのアルゴリズムよりは早く計算ができること、処理能力やメモリなどのリソースをそこまで使わなくても、かなり正確な結果を出せることです。

Aスターは次の図のように原点から目的地のルートをいろいろと試しながら最短のルートを計算します。

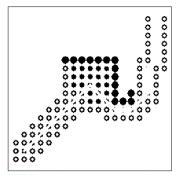



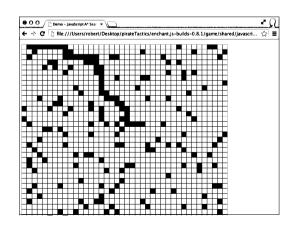
図15-1 Aスターサーチのイメージ

このAスターのアルゴリズムを数学的に説明するのはとても難しく、プログラムとして理解するのにもそれなりに経験と知識が必要なのですが、幸いにほとんどの言語やフレームワークでは、誰かがAスターアルゴリズムのライブラリを提供してくれています。私たちが使っているenchant.jsの言語JavaScriptでもAスターの実装（ライブラリ）を探すことができます。

いろいろな実装の中から今回は次のサイトにあるライブラリを選択することにしました。

https://github.com/bgrins/javascript-astar

こちらは長く使われていて実績があり、安定した動作が期待できます。今回は事前にこちらを開発環境にも含めましたので、直接ダウンロードする必要はありませんが、時間とともにライブラリは進化をしていますので、別のプロジェクトを作るなら新たにライブラリの最新版をダウンロードする必要があるかもしれません。

開発環境にあるAスターのデモ（`../game/shared/javascript-astar/demo/index.html`）をちょっと触ってみるとその動きが分かると思います。

Aスターを組み込む

ライブラリの使い方はかなりシンプルです。0と1の記号でデータ化した地図をライブラリに渡して、出発地と目的地を設定すれば最短ルートを返してくれます。簡単な使用例を見てみましょう。

```
var graph = new Graph([
    [1,1,1,1],
    [0,1,1,0],
    [0,0,1,1]
]);
var start = graph.grid[0][0];
var end = graph.grid[1][2];
var result = astar.search(graph, start, end);
```

通れないところを0、通れるところを1とする

始点

終点

最短ルートの配列が返ります

この機能は今まで作ってきた各種の機能とは違い、enchant.jsには含まれていません。そこでenchant.jsとは別にこのライブラリを読み込みましょう。久しぶりにmain.jsではなく、index.htmlを編集することにします（サンプルはchapter_15 → 27にあります）。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta http-equiv="x-ua-compatible" content="IE=Edge">
    <meta name="viewport" content="width=device-width,
    user-scalable=no">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <script type='text/javascript'
    src='../../../shared/javascript-astar/astar.js'></script>
    <script type="text/javascript"
    src="../../../../build/enchant.js"></script>
    <script type="text/javascript" src="main.js"></script>
    <style type="text/css">
        body {
            margin: 0;
            padding: 0;
```

この行を追加

```
        }
    </style>
</head>
<body bgcolor="grey">
</body>
</html>
```

<script type='text/javascript' src='../../../shared/javascript-astar/astar.js'></script>の記述によりライブラリが使えるようになったはずです。早速使ってみましょう。

まずマップデータが必要ですが、残念ながらAスターのライブラリが使っているマップデータとenchant.jsが使っているマップデータは異なります。そこでいったんGameMapの中でAスターライブラリ用のマップを生成します。これは以前の当たり判定のマップの作り方と同じです。


```
var mapSearchData = [];
for(var j=0; j < this.mapHeight; j++) {
    mapSearchData[j] = [];
    for(var i=0; i < this.mapWidth; i++) {
        if (mapData[j][i] == tileTypes.riku.id ||
        mapData[j][i] == tileTypes.iwa.id) {
            mapSearchData[j].push(0);
        } else {
            mapSearchData[j].push(1);
        }
    }
}
this.tiles.searchData = mapSearchData
```

すべてのマスを順に調べる

岩か陸であれば

0を返す

そして、このデータを使って最短経路を返すメソッドをGameMapに追加します（サンプルはchapter_15 → 28にあります）。

```
getPath: function(i,j, targetI, targetJ) {
    var start = this.searchGraph.grid[j][i];
    var end = this.searchGraph.grid[targetJ][targetI];
    var path = astar.search(this.searchGraph, start, end);
    return path;
},
```

ここでAスターライブラリを使用

drawMovementRangeの中でAスターを呼び出すだけで、それ以外の部分は今までの実装を変える必要はありません。とはいえAスターの計算は多少重いので、一部は今までの距離のチェックを残し、無駄な処理は最小限に抑えます。

移動範囲を正確にする

今までの移動可能範囲は距離だけを見ていました。今後は距離の範囲内にあるマスの最短経路を計算して、経路が移動力以内であれば移動可能にします。

```
drawMovementRange: function() {
    中略
    if (!this.outOfBorders(targetI, targetJ)) {
        if (this.getManhattanDistance(this.activeFune.i,
        this.activeFune.j, targetI, targetJ) <= this.activeFune.
        getMovement()) {
            var path = this.getPath(this.activeFune.i, this.activeFune.j,
            targetI, targetJ);
            if (path.length <= this.activeFune.getMovement()) {
                移動可能な場合の処理
            }
        }
    }
    中略
```

移動先のXのマーカーもこれに従って色を変えます

```
ontouchupdate: function(params) {
    中略
    if (this.getManhattanDistance(this.activeFune.i, this.activeFune.j,
    tile.i, tile.j) <= this.activeFune.getMovement()) {
        var path = this.getPath(this.activeFune.i, this.activeFune.j,
        tile.i, tile.j);
        if (path.length <= this.activeFune.getMovement()) {
                移動可能な場合の処理
```

15

```
        }
      中略
    },
```

上のコードは表示だけですので、GameMapの実際に移動する処理の部分についても対応させる必要があります。

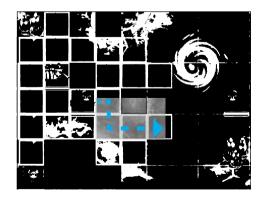
```
    ontouchend:function(params) {
      中略
        if (this.getManhattanDistance(this.activeFune.i, this.activeFune.j,
        tile.i, tile.j) <= this.activeFune.getMovement()) {
            var path = this.getPath(this.activeFune.i, this.activeFune.j,
            tile.i, tile.j);          Aスターで最短距離をもとめる
            if (path.length <= this.activeFune.getMovement()) {
                var self = this;
                utils.beginUIShield();
                self.moveFune(self.activeFune, tile.i, tile.j, function() {
                    self.controller.endTurn();
                });
            }
        }
      中略
    },
```

この辺までできると、ルール上は正確な動作になりますが、見た目のうえではまだユニットが岩を突き抜けて動きます。ユニットの移動は実際の経路に沿って動かすようにしましょう。

正しい経路で動くようにする

前に使ったTimeline (tl) の機能を`moveFune`に組み込み、正確なルートに沿って動くようにしましょう。以前は原点から目的地に一気に動くようになっていましたが、これからは正確な経路に沿って、目的地までひとマスひとマス移動を繰り返して進みます。

```
moveFune: function(fune, path, onEnd) {    ◀ 新しいmoveFuncでは船、経路を引数にする
    if (path.length > 0) {
        var nextMasu = path.shift();    ◀ 次のマスの座標を取り出す
        var i = nextMasu.y;
        var j = nextMasu.x;

        fune.i = i;
        fune.j = j;

        var position      = this.getMapPositionAtTile(i, j);
        var worldPosition = this.toWorldSpace(position.localX,
        position.localY);

        worldPosition.x -= 16;
        worldPosition.y -= 32;

        var self = this;
                                          ◀ Timelineで移動
        fune.tl.moveTo(worldPosition.x, worldPosition.y, 10,
        enchant.Easing.QUAD _ EASEINOUT).then(function(){
            self.moveFune(fune, path, onEnd);    ◀ もう一度moveFuncを呼び出して
                                                     処理を繰り返す
        });
    } else {
        onEnd();
    }
},
```

これでちゃんとしたルートに沿って移動するようになりました。

海の種類による変化

海のマスには最初から3種類ありました、「荒い」「普通」「浅い」の3つです。いままではこの海の種類は移動に影響しませんでした。幸いにも採用したAスターのライブラリには地形の移動コストを計算する機能があります。ここでは、それを活かしてそれぞれの地形に異なるコストを付けましょう。

ルールはシンプルに浅い海と荒い海の移動コストを２にします、普通の海は今までどおりコスト1にします。Aスターのマップを生成している部分のコードをすこし変更しましょう。

探索用のデータ

陸と岩の場合

通れない

荒い海の場合コスト2

浅い海の場合コスト2

```
var mapSearchData = [];

for(var j=0; j < this.mapHeight; j++) {
    mapSearchData[j] = [];
    for(var i=0; i < this.mapWidth; i++) {
        if (mapData[j][i] == tileTypes.riku.id ||
        mapData[j][i] == tileTypes.iwa.id) {
            mapSearchData[j].push(0);
        } else {
            if (mapData[j][i] == tileTypes.arai.id) {
                mapSearchData[j].push(2);
            } else if (mapData[j][i] == tileTypes.asai.id) {
                mapSearchData[j].push(2);
            } else {
                mapSearchData[j].push(1);
            }
        }
    }
}

this.searchGraph = new Graph(mapSearchData);
```

上のデータで移動コストを考慮した最短経路が取得できるはずです。現状では単純に経路を長さで計算していましたが、今後は移動に必要なコスト数で計算するようにgetPathを変更しましょう（サンプルはchapter_15 → 28bにあります）。

```
getPath: function(i,j, targetI, targetJ) {
```

```
        var start = this.searchGraph.grid[j][i];
        var end   = this.searchGraph.grid[targetJ][targetI];
        var path = astar.search(this.searchGraph, start, end);
        path.cost = 0;
        for(var i=0; i<path.length;i++){
            path.cost += path[i].getCost();   ← 経路に従って移動コストを足していく
        }
        return path;
    },
```

このように変更したら、今度はpathを利用しているすべての箇所を次のように変更しましょう。

```
if (path.cost <= this.activeFune.getMovement())
```

これで移動距離ではなく、移動コストによる最短距離が計算できるようになりました。

移動コストに応じたスピードの変化

移動コストが計算できるようになったので、マス上の移動スピードがコストに応じて変わるようにして、ちょっと爽快感をアップしましょう。コストが高いほど、移動が遅くなります（サンプルはchapter_15→29にあります）。

```
    moveFune: function(fune, path, onEnd) {
        if (path.length > 0) {
            var nextMasu = path.shift();
            var i = nextMasu.y;
            var j = nextMasu.x;
            var cost = nextMasu.getCost()

            中略
                                                    移動速度にコストを反映させる
            var self = this;
            fune.tl.moveTo(worldPosition.x, worldPosition.y, 10 *cost,
            enchant.Easing.QUAD _ EASEINOUT).then(function(){
                self.moveFune(fune, path, onEnd);
            });
            中略
```

船の種類によって変化を付けよう

いちおうこれでユニットの動きも正確になりましたが、さらに船の種類による移動コストの変化を作り込んでみましょう。

ゲームの船の種類は4種類です。速い船は軽いので浅い海でも速く移動できるようにします、しかし、同じ理由で荒い海には適さないので移動コストを2から3に上げましょう。

防御系の船は頑丈なので荒い海でも問題なく通行できるようにしましょう（コスト1）。でもその代わりに浅い海は通行しにくくなります（コスト3）。ほかの船は通常どおりの1と2のコストを採用します。それぞれの移動タイプと地形の関係を表にまとめました。

▼ 表15.1 船の種類と地形による移動コスト

	通常	スピード系（Light）	防御系（Heavy）
海	1	1	1
荒	2	3	1
浅	2	1	3

このように移動パターンは3つあるので、それぞれの移動コストのマップデータを生成します。

探索用のデータ

```
var mapSearchData = [];
var mapSearchDataLight  = [];
var mapSearchDataHeavy  = [];

for(var j=0; j < this.mapHeight; j++) {
    mapSearchData[j] = [];
    mapSearchDataLight[j] = [];
    mapSearchDataHeavy[j] = [];
    for(var i=0; i < this.mapWidth; i++) {
        if (mapData[j][i] == tileTypes.riku.id ||
        mapData[j][i] == tileTypes.iwa.id) {
            mapSearchData[j].push(0);
            mapSearchDataLight[j].push(0);
            mapSearchDataHeavy[j].push(0);
```

3種類のマップデータを作る

```
            } else {
                if (mapData[j][i] == tileTypes.arai.id) {
                    mapSearchData[j].push(2);
                    mapSearchDataLight[j].push(3);
                    mapSearchDataHeavy[j].push(1);
                } else if (mapData[j][i] == tileTypes.asai.id) {
                    mapSearchData[j].push(2);
                    mapSearchDataLight[j].push(1);
                    mapSearchDataHeavy[j].push(3);
                } else {
                    mapSearchData[j].push(1);
                    mapSearchDataLight[j].push(1);
                    mapSearchDataHeavy[j].push(1);
                }
            }
        }
    }

    this.searchGraph = new Graph(mapSearchData);
    this.searchGraphLight = new Graph(mapSearchDataLight);
    this.searchGraphHeavy = new Graph(mapSearchDataHeavy);
```

3種のマップデータにそれぞれのコストを割当てます

getPathでは船ごとに異なるマップデータを利用して経路を計算するようにします。

```
    getPath: function(fune, i,j, targetI, targetJ) {
        var path;
        if (fune.moveType == "normal") {
            var start = this.searchGraph.grid[j][i];
            var end   = this.searchGraph.grid[targetJ][targetI];
            path = astar.search(this.searchGraph, start, end);
        }
        if (fune.moveType == "light") {
            var start = this.searchGraphLight.grid[j][i];
            var end   = this.searchGraphLight.grid[targetJ][targetI];
            path = astar.search(this.searchGraphLight, start, end);
        }
        if (fune.moveType == "heavy") {
            var start = this.searchGraphHeavy.grid[j][i];
            var end   = this.searchGraphHeavy.grid[targetJ][targetI];
```

通常

スピード系

防御系

```
        path = astar.search(this.searchGraphHeavy, start, end);
    }

    path.cost = 0;
    for(var i=0; i<path.length;i++){
        path.cost += path[i].getCost(); // コストを合計する
    }
    return path;
},
```

あとはそれぞれの船が採用している移動パターンを定義します。

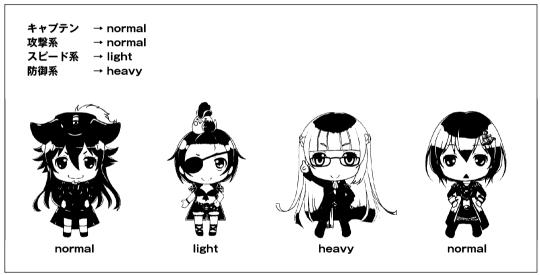



図15-2 ユニットと移動パターンの対応

BaseFuneはmoveTypeをnormalに設定します。

```
var BaseFune = Class.create(Group, {
    initialize: function(id, stats) {
        中略
        this.moveType    = "normal";
    },
    中略
```

スピード系の船の定義はlightにして、

```
var HayaiFune = Class.create(BaseFune, {
    initialize: function(id) {
        中略
        this.moveType    = "light";
        中略
    },
    中略
```


防御系の船の定義をheavyにします。

```
var KataiFune = Class.create(BaseFune, {
    initialize: function(id) {
        中略
        this.moveType    = "heavy";
        中略
    },
    中略
```

これでシミュレーションゲームらしく地形が影響するようになり、ゲームの戦略性と奥深さが一段とアップをしました！

Chapter 16

必殺技を作る

本章で勉強すること

- キャラクターの個性を光らせる必殺技を作り込もう
- まず基本的な機能を作り込んでそれを拡張させていきます
- 継続の機能を利用したいときはプロトタイプが便利です

各ユニットのパラメータには違いがあり、ユニットによる地形の特性もゲームの中に再現されているので今のままでもいちおうの個性は出ています。でもそれにプラスαで各船に独自の必殺技を実装しましょう。これでユニットごとの個性が一層際立ちます！

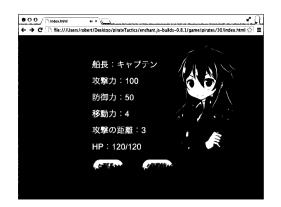
必殺技はゲームの面白さと奥深さに直接繋がります。しかし、必殺技のような機能は基本のゲームルールの例外になりますので、実装はパート3の最後に持ってきました。

考えられる必殺技はいくらでもありますが、これから4つの必殺技を作り込みます。それぞれが異なる特徴をもつ必殺技ですので、今後はこれらを元にオリジナルな技も作れると思います。

必殺技の使い方

このゲームで必殺技を実行するのはステータスウィンドウからにしますので、まずスキル発動用のボタンを新規に追加しましょう（サンプルはchapter_16 → 30にあります）。

```
initialize: function(fune) {
    中略
    var skillBtnSprite = new Sprite(128, 64);   ← スキル発動ボタンのスプライトを追加
    skillBtnSprite.image = game.assets[uiSkillBtnSprite];

    中略
    windowGroup.tl.scaleTo(1, 10, enchant.Easing.ELASTIC _ EASEOUT).
    then(function() {
        中略                                   ↓ ボタンを押した際のイベントリスナーを追加
        skillBtnSprite.addEventListener(enchant.Event.TOUCH _ END,
        function(params) {
            中略
            windowSprite.tl.fadeTo(0, 5).then(function() {
                game.popScene();
                fune.player.controller.sndManager.playFX(sndClick);
                if (self.onSkill) {
                    self.onSkill()
                }
            });
        });
    });
},
```

これでステータスウィンドウから必殺技（スキル）の実行ができるはずです。

必殺技の共通部分を作る

必殺技の基本のルールは、原則として次のようにします。

1. 1つのユニットは1つのミッションで必殺技を1回だけ使用できる
2. 必殺技を使うと自分のターンが終わる

最初はこの2つのルールを守りながら、実際には何も起きない必殺技を`BaseFune`に追加することにしましょう。それを拡張してそれぞれの船特有の必殺技を作り込みます（サンプルはchapter_16 → 31にあります）。

```
var BaseFune = Class.create(Group, {
    中略
    activateSkill: function(onEnd) {
        self.usedSkill = true;
        utils.beginUIShield();
        this.processSkill(onEnd)
    },
```

1.一つのユニットは一つのミッションで必殺技を1回だけ使用できる

スキル実行メソッド

`activateSkill`を実行しても実際には何も起きません。しかし、それぞれの船でこのクラスを継承することにより、各種の必殺技が実行できるようになります。

```
processSkill: function(onEnd) {
    onEnd();
},
```

これらをステータスウィンドウのスキルボタンに連動させます。技が発動したらターンを進めます。

```
ontouchend: function(params) {
    if (this.player.isActive()) {
        if (this.player.getActiveFune() == this) {
            var popup = new StatusWindow(this);
            中略
            var self = this;
            popup.onSkill = function() {
                if (self.usedSkill == false) {
                    self.activateSkill(function() {
                        self.player.controller.endTurn();
                    })
                }
            }
    中略
```

2.必殺技を使うと自分のターンが終る

必殺技の基盤となる実装はできましたので、これからは小さなものから大がかりなものまで技を実装していきましょう。

キャプテンの技

キャプテンには、みんなをサポートするように回復技を付けます。

この技の実装はとても簡単です。自分の船すべてに体力の半分を計算して回復させます。


```
var CaptainFune = Class.create(BaseFune, {
    中 略
    processSkill: function(onEnd) {
        var count = this.player.getFuneCount();
        for (var i=0; i < count; i++) {
            var fune = this.player.getFune(i);
            var toHeal = Math.ceil(fune.getHPMax() /2);
            fune.healDamage(toHeal);
        }
        onEnd();
    },
});
```

体力の半分（小数点以下切捨）を計算

回復

攻撃系の技

攻撃系はキャプテンの技に近いですが、距離もチェックが必要です。そしてダメージの表現と沈没のチェックが必要となります。


```
var KougekiFune = Class.create(BaseFune, {
    中略
    processSkill: function(onEnd) {
        var damage = this.stats.attack;
        var count = this.player.controller.getNonActivePlayer().
        getFuneCount();
        for (var i=0; i < count; i++) {   ◀ 敵のすべての船に対し
            var fune = this.player.controller.getNonActivePlayer().
            getFune(i);
            if (this.withinRange(fune.i, fune.j)) {   ◀ 攻撃可能距離にあれば
                var afterHp = fune.takeDamage(damage);
                var explosion = new Explosion();      ▲ ダメージ
                explosion.x = fune.x +32;
                explosion.y = fune.y +32;
```

```
                    this.player.controller.sndManager.playFX(sndExplosion);
                    game.currentScene.addChild(explosion);

                    if (afterHp <= 0) {  ◀ HPが0の場合は沈没
                        fune.sinkShip();
                    }
                }
            }
            onEnd();
        },
});
```

防御系の技

技名：アイロンシールド

説明：次に動くまで
無敵状況に入る

効果

1. ユニットに無敵フラグを追加する
2. 技を使うと無敵をtrueにする
3. 船が攻撃されてもダメージを受けない
4. takeDamageのなかで無敵の状態を確認して、無敵であればダメージは無視する
5. 一回攻撃されたら無敵状態が消える

防御系の必殺技のprocessSkillの実態は単純ですが、無敵状況を再現するためにattackFuneとtakeDamageを変更する必要があります。

ここではJavaScript特有の機能であるプロトタイプを使ってみましょう。

```
var KataiFune = Class.create(BaseFune, {
    initialize: function(id) {
        中略
        this.indestructible = false;  ◀ 無敵フラグを作る
    },
    中略
    processSkill: function(onEnd) {  ◀ processSkillは無敵フラグをオンにするだけ
        this.indestructible = true;
        onEnd();
    },

    attackFune: function(otherFune) {
        this.indestructible = false;
        BaseFune.prototype.attackFune.call(this, otherFune);
    },

    takeDamage: function(damage) {  ◀ 防御系のためにtakeDamageを再定義
        if (this.indestructible) {
            this.indestructible = false;  ◀ スキル発動時の処理
            return this.getHP()
        } else {
            return BaseFune.prototype.takeDamage.call(this, damage);
        }                       ▲ 元のtakeDamageの処理を呼び出す
    },
});
```

JavaScriptはオブジェクト指向のプログラミング言語ですが、JavaやC++と異なり、純粋なクラスではなく、**プロトタイプ**（prototype）を使ってオブジェクト指向の機能を実現しています。

プロトタイプは、直訳すると「試作品」または「見本」になります。JavaScriptのプロトタイプの場合どちらかというと「見本」という訳のほうが合っているかもしれません。

JavaScriptのすべてのオブジェクトには、プロトタイプ（見本）が用意されています。たとえば上記のコードのKataiFuneはBaseFuneをプロトタイプとして作られています。オブジェクトに定義してないデータやメソッドがあっても、自分のプロトタイプにあれば参照が可能です。

コードの中ではattackFuneとtakeDamageを新たに定義していますが、元になったオブジェクト（プロトタイプ）の機能を、

```
BaseFune.prototype.takeDamage.call(this, damage);
```

というコードで呼び出しています。こうすれば、元のクラスの仕様と同じ部分はいちいち書き直す必要はありません。

今までクラスのinitializeの中で親クラスのコンストラクタを呼んでいましたが、この部分でもプロトタイプの仕組みは使われています。

素早さ系の技

これは船の実態に影響しない技です、どちらかというとゲームのルールを大幅に変えています（チートレベル）。そのため船クラス内の実装は簡単ですが、実際の処理はGameManagerを変更します。

```
var HayaiFune = Class.create(BaseFune, {
    中略
    processSkill: function(onEnd) {
        this.player.controller.getFreeTurns(this.player, 2);
        onEnd();
    },
});
```

getFreeTurnsで誰を何ターン飛ばすかを決めます。このメソッドをGameManagerに加えます。

```
getFreeTurns: function(player, turns) {
    this.skipper = player;
    this.skipTurns = turns;
}
```

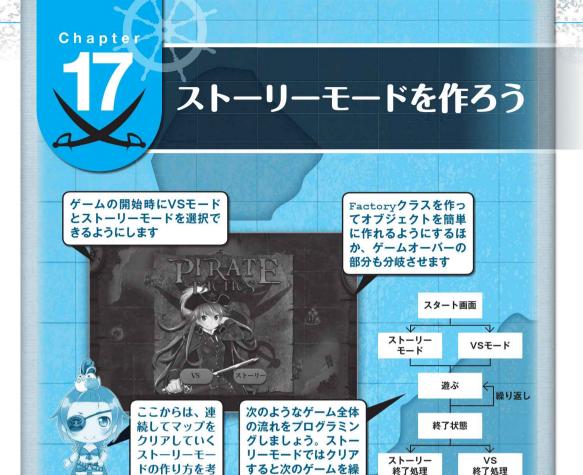
ここで設定したプロパティをstartTurn内で確認して実際にターンを飛ばします。

```
startTurn: function() {
    var player = this.getActivePlayer();
    if (this.skipTurns) {    skipTurnsがあるときは
        if (player != this.skipper) {
            this.skipTurns--;
            return this.endTurn();    ターン終る
        }
    }
    player.setActive(true);
    this.updateTurn();
},
```

これで各キャラクターそれぞれの個性的な必殺技が実装できました。こまで来るとプレイヤー同士が対戦するマルチプレイヤー型のゲームとしてほぼ完成したことになります。

しかし、これだけではまだCPUを対戦相手とするシングルプレイヤーゲームとしては完成していません。パート4ではCPU対戦機能を作るとともに、ゲームのさまざまな仕上げを行いたいと思います。

Part

4 コンピュータ対戦への道

本章で勉強すること

- ゲームストーリーモードを加えてシミュレーションゲームらしくしよう
- まずスタート画面のメニューを考えてみよう
- ステージごとの登場ユニットをデータ化してみよう
- Factoryクラスを使ってデータからオブジェクトを作る方法を紹介
- ゲームの開始とゲームオーバーを作り込んで完成度を上げよう

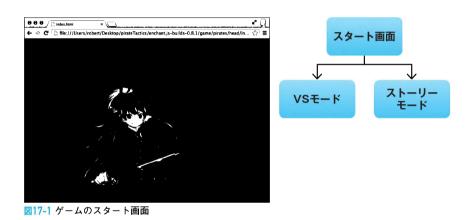
今まで私たちが作ってきたゲームは人間同士が交互に操作を繰り返すチェスのような対戦用ゲームになっていました。しかし、いつでも対戦相手がいるとは限りませんので、ここからは一人でも遊べる仕組みを追加していきたいと思います。

スタート画面

このパートではゲームのシングルプレイモードとそのための画面遷移を用意します。最終的には19章で解説する人工知能（AI）を実装すると、シングルプレイのゲームモードが実現します。

本ゲームは2つのモードで遊べるようにしましょう。ひとつ目は今までどおりの対戦モードです。このモードでは双方のプレイヤーが（たとえ対戦相手が人工知能のCPUであっても）同じような4隻のチームを持って勝負します。ふたつ目はストーリーモードです。このモードでは物語の流れに従って、プレイヤーがステージを進みます。対戦相手はCPUで、ステージで勝利すると、新たなステージに進む方式です。ゲームの進み具合を保存する方法については18章で説明します。

ゲームのスタートからの画面の流れは次のようになります。

図17-1 ゲームのスタート画面

では最初にスタート画面を作ります。この画面は基本的に以前作ったステータス画面と同じような作りになります（サンプルはchapter_17 → 32にあります）。

```
var StartScreen = Class.create(Scene, {
    initialize: function(gameManager) {
        中略
        versusBtnSprite.addEventListener(enchant.Event.TOUCH _ END,
        function(params) {
            中略
            windowSprite.tl.fadeTo(0, 5).then(function() {
                gameManager.beginVersusGame();
                gameManager.sndManager.playFX(sndClick);
```

VSモードの処理

```
                    game.popScene();
                });
            });
            中略
            campaignBtnSprite.addEventListener(enchant.Event.TOUCH _ END,
            function(params) {
                中略
                windowSprite.tl.fadeTo(0, 5).then(function() {
                    gameManager.beginCampaignGame();
                    gameManager.sndManager.playFX(sndClick);
                    game.popScene();
                });
            });
        },
    })
```

ストーリーモードの処理

押したボタンversusBtnSpriteとcampaignBtnSpriteによってそれぞれのモードでゲームを開始するStartScreenを作りました。これでスタート画面から2つのモードに入れるようになります。game.onloadでは最初にこの画面が表示されるようになっています。

```
game.onload = function(){
    var sceneGameMain = new Scene();

    var manager = new GameManager();

    var mapDisplayData = [
        中略
    ];

    var map = new GameMap(sceneGameMain, mapDisplayData);
    manager.setMap(map);

    var frameUI = new FrameUI(sceneGameMain);
    manager.setFrameUI(frameUI);
    game.pushScene(sceneGameMain);
```

GameManagerの作成

マスのデータの作成

ゲームにシーンを追加

```
        new StartScreen(manager);  ◀ モード選択画面を表示
    };

    game.start();
};
```

VSモードの実装

VSモードの部分はここまで作ってきた2プレイヤーのゲームのままですので、あまり変更はありません。メソッドの名前などを変更しています（サンプルはchapter_17 → 32にあります）。

```
beginVersusGame: function() {
    this.mode = "versus";

    船の初期の位置
    var startPositions = {
        player1: [
            {i: 0, j: 8}, {i: 0, j: 6}, {i: 1, j: 7}, {i: 2, j: 8}
        ],
        player2: [
            {i: 12, j: 0}, {i: 10, j: 0}, {i: 11, j: 1}, {i: 12, j: 2}
        ],
    }
    this.setStartPositions(startPositions);

    プレイヤー1
    var player1 = new GamePlayer(1, {name:"プレイヤー1"});
    this.addPlayer(player1);

    プレイヤー1に船を4つあげよう
    player1.addFune(new CaptainFune(1));
    player1.addFune(new HayaiFune(2));
    player1.addFune(new KataiFune(3));
    player1.addFune(new KougekiFune(4));

    this.placePlayerShips(player1);
```

17

```
    プレイヤー2
    var player2 = new GamePlayer(2, {name:"プレイヤー2"});
    this.addPlayer(player2);
    プレイヤー2に船を4つあげよう
    player2.addFune(new CaptainFune(1));
    player2.addFune(new HayaiFune(2));
    player2.addFune(new KataiFune(3));
    player2.addFune(new KougekiFune(4));

    this.placePlayerShips(player2);

    this.sndManager.playBGM();
    this.startTurn();
},
```

ストーリーモードの実装

ストーリーモードは今までのゲームと異なるところが多いので、新しく開発する部分もかなりあります。最初は、新しいモードの入り口となるメソッドbeginCampaignGameを用意します。プレイヤー1は前のbeginVersusGameとあまり変わりませんが、プレイヤー2の追加と船の設定はsetupStageで行い、ステージに応じて船のラインナップを変化させられるようにします。

17

```
beginCampaignGame: function(stage) {
    this.mode = "campaign";
    プレイヤー1
    var player1 = new GamePlayer(1, {name:"プレイヤー1"});
    this.addPlayer(player1);
    プレイヤー1に船を4つあげよう
    player1.addFune(new CaptainFune(1));
    player1.addFune(new HayaiFune(2));
    player1.addFune(new KataiFune(3));
    player1.addFune(new KougekiFune(4));
```

```
    船の初期の位置
    var startPositions = {
        player1: [
            {i: 0, j: 8}, {i: 0, j: 6}, {i: 1, j: 7}, {i: 2, j: 8}
        ],
    }
    this.setStartPositions(startPositions);
    this.placePlayerShips(player1);

    if (stage) {
        this.setupStage(stage); ◀ステージセットアップのためのメソッドを実行
    } else {
        var player1 = new GamePlayer(2, {name:"敵"});
        this.addPlayer(player1);
        this.setupStage(1); ◀ステージデータがない場合はステージ1から
    }

    this.sndManager.playBGM();
    this.startTurn();
},
```

setupStageの中身は次のようになります。ちょっと変わったところとしては**startPositions**のそれぞれの要素**type**に船の種類と初めの位置が文字列で入っていることでしょうか。この文字列を**funeFactory**のメソッドに渡すと、適切な船のオブジェクトが作られるようになっています。

17

```
setupStage: function(stageId) {
    this.stageId = stageId;

    var StageData = [
        {
            startPositions: [
                {type: "captain", i: 12, j: 0},
            ]
        },
        {
            startPositions: [
                {type: "captain", i: 12, j: 0},
```

```
                    {type: "hayai", i: 12, j: 2},
                ]
            },
        ];

        var player2 = this.getPlayer(2);

        var stageIndex = (stageId-1) % StageData.length;
        var stageData = StageData[stageIndex];
        for (var i=0; i< stageData.startPositions.length; i++) {
            var entry = stageData.startPositions[i];

            var fune = this.funeFactory(entry.type);
            fune.originX = 32+16;
            fune.scaleX = -1;
            player2.addFune(fune);
            this.map.addChild(fune);
            this.map.positionFune(fune, entry.i, entry.j);
        }
    },
```

今は2つのステージを繰り返して遊べる

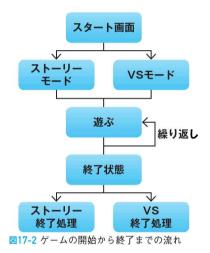
上で使われている**funeFactory**は名前のとおり工場（Factory）のように、与えられたデータによってそれぞれの船を作ってくれます。**factory**クラスは地味ですが、ゲームのプログラミングではとてもよく見かける方法です。**factory**とデータを使って異なるステージなどを楽に作ることができます。**funeFactory**の中も見てみましょう。

```
    funeFactory: function(name) {
        switch(name) {
            case "captain":
                return new CaptainFune(1);
            case "hayai":
                return new HayaiFune(2);
            case "katai":
                return new KataiFune(3);
            case "kougeki":
                return new KougekiFune(4);
        }
    },
```

与えられた文字列によって異なる種類の船を生成します。どのゲームでも基本的にはこのようなデータを元に各ステージ用のオブジェクト生成する方式が一般的です。ひとつひとつのステージに別のクラスやデータを用意したりすることはあまりありません。

分岐によるゲーム終了の変更

ゲームの開始は整理できましたので今度はゲーム終了まわりを整理しましょう。さきほどのスタートのフローに合わせ、終了のフローを含めたゲームの流れは次のようになります。

図17-2 ゲームの開始から終了までの流れ

今までのendTurnの終了条件の判定の部分を変更して、2つのモードの終了処理を加えます。

```
endTurn: function() {
    var player = this.getActivePlayer();
    player.setActive(false);

    var winner = this.getWinner();
    if (winner) {
        var self = this;
        setTimeout(function(){
            if (self.mode == "versus") {    ◀ VSモード
                self.versusOver(winner);
            } else if (self.mode == "campaign") {    ◀ ストーリーモード
```

17

```
                    self.campaignOver(winner);
                }
            }, 1000);
```

self.modeが"versus"か"campaign"によって終了処理を切り替えるようになっています。

VSモードの終了画面

スタートと同じように対戦モードの終了も大きな変更はありません。モードが分かれたのでそれに合わせてすこし整理しました。

図17-3 対戦モードの流れ

```
versusOver: function(winner) {
    var touchable = new ShieldWindow(this);
    utils.beginUIShield();

    touchable.onTouch = function() {
        location.reload();
    };

    var playerBanner = new Sprite(512, 256);
    if (winner.id == 1) {
        playerBanner.image = game.assets[uiPlayerBanner1];
    } else if (winner.id == 2) {
        playerBanner.image = game.assets[uiPlayerBanner2];
```

勝ったプレイヤー名を表示

17

```
        }
        中略

        var self = this;
        playerBanner.tl.fadeIn(20).delay(30).fadeOut(10).then(function() {
            game.currentScene.removeChild(playerBanner);

            var resultBanner = new Sprite(512, 256);
            中略

            resultBanner.tl.fadeIn(20).then(function(){
                utils.endUIShield();
            });
        });
    },
```

ストーリーモード の終了画面

ストーリーモードは、プレイヤー１が勝ったか負けたかによって、２つの処理に分けます。

図17-4 ストーリーモード

プレイヤー 1が勝ったときは次のステージに進みます。

```
campaignOver: function(winner) {
    var touchable = new ShieldWindow(this);
    utils.beginUIShield();

    var playerBanner = new Sprite(512, 256);
    playerBanner.image = game.assets[uiPlayerBanner1];

    中略

    var self = this;
    playerBanner.tl.fadeIn(20).delay(30).fadeOut(10).then(function() {    ← プレイヤー1を表示
        game.currentScene.removeChild(playerBanner);

        var resultBanner = new Sprite(512, 256);
        if (winner.id == 1) {
            resultBanner.image = game.assets[uiWin];    ← 勝利
            touchable.onTouch = function() {
                self.refreshPlayer(winner);
                self.setupStage(self.stageId +1);
                self.startTurn();
            };
```

refreshPlayerの部分は次のようになります。

```
refreshPlayer: function(player) {
    for (var funeIndex = 0; funeIndex < player.getFuneCount(); funeIndex++) {
        var fune = player.getFune(funeIndex);
        var recoverHp = fune.getHPMax();
        fune.healDamage(recoverHp);    ← 各船のダメージ回復
        fune.refreshSkill();
    }
    this.placePlayerShips(player);
    this.turnCounter = 0;
    this.skipTurns = 0;
},
```

17

プレイヤー 1が負けたら、対戦モードと同じようにreloadで最初のスタート画面に戻ります。

```
campaignOver: function(winner) {
    中略
        } else if (winner.id == 2) {     ◀敗北
            resultBanner.image = game.assets[uiLose];
            touchable.onTouch = function() {
                location.reload();     ◀リロード
            };
        }
        中略

        resultBanner.tl.fadeIn(20).then(function(){     ◀バナーを表示
            utils.endUIShield();
        });
    });
},
```

VSモードはこれでほぼ完成しました。でもストーリーモードにはまだいろいろとやることが残っています。次の章ではストーリーモードの進み具合を保存する方法を考えましょう。

本章で勉強すること

- 現在の状態ではブラウザをリロードするとゲームがすべて初期化されてしまう
- リロードしてもステージの状態が保存されるようにしたい
- データの保存にはブラウザの違いを吸収してくれるjStorageフレームワークを利用しましょう
- ステージが始まるたびにステータスを自動で保存してくれる仕組みを作り込もう
- 音量のデータも自動で保存させるようにしよう

18

ストーリーモードはそれぞれのステージが進むにしたがって対戦相手のデータが変化していきます。そのためゲームの進み具合を何かの方法で保存しておく必要があります。こうしておかないと、ブラウザからゲームをリロードするたびに、始めの状態に戻ってしまいます。それぞれのステージを作り込む前にゲ

ームの進み具合を保存する仕組みを作っておきましょう。

データ保存のための仕組み「jStorage」

サウンドのときと同じく、データの保存はブラウザごとに方式が異なります。そのため、ブラウザの違いを吸収してくれるデータ保存のためのフレームワークを採用したいと思います。

jStorage（https://github.com/andris9/jStorage）はブラウザのデータ操作を簡単にしてくれるフレームワークで、代表的なブラウザにはほとんど対応しています。Webサイトを見るとjStorageにはいろいろな機能がありますが。ここで使うのは基本的に次の機能だけです。

set(key, value[, options])
データを保存する。keyはデータの名前、valueは保存したいデータ。データは文字列、数字または一部のオブジェクトが利用できる

get(key[, default])
データを読み込む。keyはデータの名前、defaultの部分にはデータがなかったときに返すデフォルトのデータを指定できる

deleteKey(key)
特定のkeyに紐付けられたデータを消す

音量を記録する

ゲームデータの保存の仕組み作る前に、シンプルな課題でjStorageの機能を見てみましょう。今回のゲームでは音量の設定はできましたが、ゲームをロードするたびに毎回設定し直す必要がありました。それではちょっと不親切ですので、プレイヤーが選んだ音量の設定を自動的に保存するように仕様を変更してみましょう。

まず、jStorageをmain.jsの中で使えるようにするために、index.htmlを編集して次のようにしてください（サンプルはchapter_18 → 34にあります）。

```
<script  type='text/javascript'  src='../../../shared/javascript-astar/astar.
js'></script>
<script type='text/javascript' src='../../../shared/jStorage-master/jstorage.
```

jStorageの読み込み

```
js'></script>

<script type="text/javascript" src="../../../../build/enchant.js"></script>
```

記述は、aster.jsを利用するときと同じです。あらかじめjstorage.jsを読み込んでおくことで、main.js内で利用可能になります。

準備ができたら、main.js内のSoundManagerのコードを変更します。

```
var SoundManager = Class.create(Sprite, {
    initialize: function() {
        Sprite.call(this, 1,1);
        this.volume = $.jStorage.get("sound volume", 0.5);
        this.bgmPlaying = false;
    },

    中略
    volumeUp: function() {
        中略
        $.jStorage.set("sound volume", this.volume);  ◀ボリュームを上げたら保存
        中略
    },

    volumeDown: function() {
        中略
        $.jStorage.set("sound volume", this.volume);  ◀ボリュームを下げたら保存
        中略
    },
```

18

ご覧のとおりとてもシンプルです。`jStorage.get("sound volume", 0.5);`の部分で、jStorageからボリュームデータを取り出して（データがなければ0.5を設定）音量として設定しています。

ちょっと不思議なのは`$.jStorage`というオブジェクトの名前でしょうか。頭にある`$.`が妙な感じに見えますが、JavaScriptの変数名では_や$が数字や文字のように使えます（言語によってはこのような特殊な文字が使えないこともあります）。$を付けるのはjQueryという汎用的なJavaScriptのフレームワーク（ゲーム用ではなくWebページ用として広く使われています）の一般的な命名規則です。jStorageもjQueryを意識しているので、同じ命名規則を採用して`$.jStorage`になっているのです。

ゲームのセーブデータの保存

　では本章の本題に入りましょう。ゲームの進捗データを保存する仕組みを作ります。
　新しいステージをセットアップするときに最新の進捗状態を記録することにします。今回はステージの始まりのときだけ保存します。ステージ途中の状況を保存するのも同じ方式でできますが、かなり手間がかかりますので、今回はシンプルにステージ単位で保存する方式にしましょう（サンプルはchapter_18→35にあります）。

```
setupStage: function(stageId) {
    this.stageId = stageId;

    var saveData = {
        stageId: stageId,          ← 保存用のオブジェクトを作る
        funeList: [],
    };
    var player = this.getPlayer(1);
    for (var funeIndex = 0; funeIndex < player.getFuneCount(); funeIndex++) {
        var fune = player.getFune(funeIndex);
        saveData.funeList.push(fune.getId()); ← 今回は船の種類だけを保存します
    }
    $.jStorage.set("save data", saveData); ← ここで記録
```

saveDataというオブジェクトには**stageId**に現在のステージIDと現在プレイヤーが持っている船の種類を記録することにしましょう。これでデータのセーブが自動化されました。
　ストーリーモードを始めるときにはセーブデータを読み込み、前回のステージの初期状態に戻します。

```
beginCampaignGame: function(stageId) {
    this.mode = "campaign";

    var funeList;
    var saveData = $.jStorage.get("save data"); ← ここで読み込み
    if (saveData) {
        stageId  = saveData.stageId;      ← 保存していたstageIdを使う
        funeList = saveData.funeList;     ← 保存していた船の種類を使う
```

18

```
    }
    プレイヤー1
    var player1 = new GamePlayer(1, {name:"プレイヤー1"});
    this.addPlayer(player1);

    if (funeList) {
        for (var funeIndex = 0; funeIndex < funeList.length; funeIndex++) {
            var fune = this.funeFactory(funeList[funeIndex]);
            player1.addFune(fune);
        }
    } else {  プレイヤー1に船を4つあげよう

        player1.addFune(new CaptainFune(1));
        player1.addFune(new HayaiFune(2));       funeListがなかったときの処理
        player1.addFune(new KataiFune(3));
        player1.addFune(new KougekiFune(4));
    }
```

save dataというキーでデータを引き出し**stageId**と**funeList**を展開します。

これでストーリーモードのデータを引き継ぐようになりました。ただし、現状では対戦に負けても同じステージが継続されます。本当は対戦に負けたらはじめのステージに戻るようにしたいので、最初のスタート画面周りを拡張しましょう。

ストーリーモードを選んだあと、もう1つ画面を挟んで保存データがあれば「ロードする」と「ニューゲームを始める」という選択が表示されるようにします。

18

```
var CampaignScreen = Class.create(Scene, {
    initialize: function(gameManager) {
        中略
        var saveData = $.jStorage.get("save data");  保存データがあるかどうかを確認
        if (saveData) {
            中略
            continueBtnSprite.addEventListener(enchant.Event.TOUCH_END,
```

```
                function(params) {
                    中 略
                    gameManager.beginCampaignGame(); ◀ストーリーモード開始
                    中 略
                });
            }
            中 略
            newBtnSprite.addEventListener(enchant.Event.TOUCH _ END,
            function(params) {
                $.jStorage.deleteKey("save data") ◀ニューゲームのときは保存データを消します
                中 略
                gameManager.beginCampaignGame();◀ストーリーモード開始
                中 略
            });
        },
    })
```

ストーリーモードで負けたらゲームのセーブデータは初期化します。また遊ぶときはステージ１から始まります。

```
    campaignOver: function(winner) {
        中 略
        var resultBanner = new Sprite(512, 256);
        if (winner.id == 1) {
            → ◀プレイヤー 1が勝ったときの処理
        } else if (winner.id == 2) { ◀負けたときの処理
            resultBanner.image = game.assets[uiLose];
            touchable.onTouch = function() {

                $.jStorage.deleteKey("save data");
                location.reload();
            };
        }
        中 略
    },
```

これでデータ保存の仕組みは完成しました。次はいよいよ、対戦時のロジックの実装を考えてみましょう。これを作りこむことで、コンピュータを相手に一人でプレイできる本格的なゲームになります。

本章で勉強すること

- AI（人工知能）の基礎知識を身に付けよう
- AIには強いAIと弱いAIがあるが、ゲームの場合は弱いAIに属する
- 思考を表現する方法としてステートマシンという手法を使う
- ゲームの段階を「序盤」「中盤」「終盤」と位置付けて、条件によって状態（ステート）を変化させる
- ステートごとにユニットを動かすルールを作ろう
- ついでに、対戦モードでもAIと対戦できるようにしておこう

19

第1章でも説明していますが、AI（Artificial Intelligence：人工知能）はコンピュータの歴史の初期のころから研究対象となっていました。その後、何十年かかかりましたが、現在ではチェスや将棋の人工

知能は最強の人間のプレイヤーにも勝てるところまで進化しています。

強いAI と 弱いAI

ゲームで使うAI、将棋の世界の最強のAI、医学分野における病気判定の補助、郵送物の宛先を判定してくれるAIなどはどれも**弱いAI**（Weak AI）と呼ばれるものです。これらは人間の能力の一部をソフトウェアで再現する目的で開発されています。

一方、**強いAI**(Strong AI)はSFに出てくるような人間と同じくらいの汎用性を持つAIです。映画『2001年宇宙の旅』に出てくるコンピュータHAL9000は1970～80年ごろの研究者が2000年までに実現できそうなものとして考えた強いAIでした。もちろん、実際には完全な強いAIまだしばらく実現しないと思われます。現代の強いAIは、まだ虫のレベルを不完全に再現するくらいまでにしか進化していません。

今回作るようなゲームの場合はもちろん弱いAIで十分ですから、本書ではAIの本当に深いところには触れません。

ゲームにおけるAIの目的

例外もありますが、一般的なゲームのAIは一人で遊ぶプレイヤーの対戦相手になるよう作られています。多くのゲームには何らかの戦闘（勝負）の要素が含まれます。ゲームAIは、戦闘における敵の振る舞いをより人間らしく見せるために作られます。

ゲームのAIも大きく2種類に分かれます。ひとつはチェスや将棋などのゲームにおける「強いプレイヤー」の再現を目的としたAIです。ただし、コンピュータと人間の対戦の場合は、人間側が公平さを感じるような調整も必要です。たとえば、FPS（First Person Shooting：プレイヤーの視点を再現したシューティングゲーム）の場合、NPC（Non Player Character：コンピュータに操作されているゲーム中の登場人物）は人間の能力を超えた完璧な射撃もできますが、それでは公平さが感じられません。

もうひとつのAIはRPGの敵キャラのような比較的シンプルなAIです。これらはゲームの進行上適度に敵対してきますが本気でプレイヤーを全滅させるようなことはしません。このようなAIはアドベンチャーゲームの雑魚キャラなどで使われます。ゲームに大切なのは楽しくあることです。ゲームAIは最終的にユーザーから見て知性が感じられるか、ゲームプレイが心地よいかを重視しているため、通常のAIとは手法が大きく異なります。

本式のAIである必要はありませんし、目的の範囲でチート（もともとは「覗き見」の意味で、本来見られないはずの相手のステータスを読み取るような方法）も許容されるでしょう。

この本ではこのような、ちょっとだけ強い対戦相手になるAIを作ることにします。

AIのデザイン～トップダウンとボトムアップ～

AI作成のアプローチには大きくトップダウン（Top Down）とボトムアップ（Bottom Up）の2つの手法があります。その名のとおりトップダウン式は組織の一番上のモノ（AI）が部下（ユニット）をコントロールします。対戦系のAIには戦略が必要とされるのでトップダウンアプローチをとることが多いです。

ボトムアップ式はそれぞれのユニットが自分で判断して動く方式です。これは動物や魔物のようにコミュニケーションを重視しないような敵に向いています。どのようなシナリオを再現したいかにより、AIのデザインもすこしずつ変わってきます。

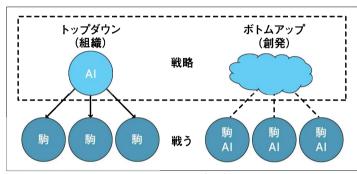



図19-1 トップダウンアプローチとボトムアップアプローチ

今回のゲームはチェスのように組織（人間の集団）が敵です。このような場合はプレイヤーの意思決定が大切ですので、トップダウン式にAIを作ります。AIのプレイヤーが判断して自分のリソース（船）を動かします。

ゲームの戦略

チェスや将棋、今回のシミュレーションゲームのようなボード形ゲームの場合、戦略は基本的に3つのフェーズに分けて考えることがよくあります。具体的には序盤（Opening）、中盤（Mid-Game）、終盤（End-Game）です。各フェーズの境目は多少曖昧ではありますが、今回はこのように分解して、ゲームに適用していくことにします。

序盤

開始状態から、AIプレイヤーは中盤に向けて、ゲームを有利に進めるため、いち早く良いポジションを狙い、敵をよけながらフィールドに展開します。

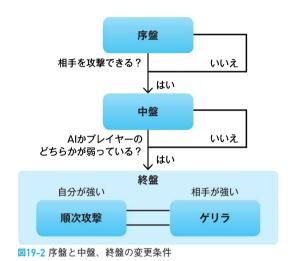
中盤

強いポジションを得たところで攻撃を始めます。ここでは自分の身を守りながらも、敵と全力で戦います。

終盤

終盤は中盤の成果により戦略が変化します。基本的に自分が有利または不利な状況に応じて戦略が決まります。自分が有利な場合は不利な側（プレイヤー）のユニットを順次攻撃していきます。自分が不利な立場にいる場合は、ゲリラ（不正規兵）のように有利な敵に対してできる限りの抵抗を続けます。

このような基本戦略をAIに適用する方法を採るとAIの作り方はずっと楽になります。AIにゲームの仕組みを全部理解させて難しい判断をさせるよりは、このような大まかな戦略を決めてからAIをデザインをすると早いです。図にしてみると次のようになります。

図19-2 序盤と中盤、終盤の変更条件

この図の内容を、そのままAIのロジックに落としましょう。

ステートマシン

ここで例を示すのは**ステートマシン**あるいは**有限状態機械**といわれる方式です。FSM（Finite State Machine）と略すこともあります。「なんか難しそう…」と思うかもしれませんが、要はプログラムの書き方の一種です。とくにFSMはゲームAIやストーリーを作るのに向いています。典型的なFSMのロジックを示してみます。

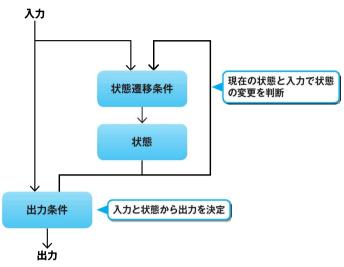



図19-3 典型的なステートマシンの遷移図

この図の場合「状態」に相当するのが、今回のゲームでの「序盤」「中盤」「終盤」です。状態が変わると、入力を判断をするルールが変わる仕組みになっています。何か入力があると現在の状態とその入力を元に状態が変化するかを判断します。そして、決定した状態と入力を元に、結果が出力されます。この仕組みを、各フェーズごとの戦略にあてはめていきましょう。

まず、基本となるクラスとして**BaseState**を作ります。**AIPlayer**が呼び出すメソッドのインターフェイスです。共通のロジックはこちらに実装していきます。

AI Playerが使う基本のインターフェイスは次の2つのメソッドになります。

```
var BaseState = Class.create({
    updateState: function() {
        中略
        return nextState;
    },
    chooseAction: function() {
        中略
        return action;
    }
})
```

今回使う戦略の状態を更新する。このままの状態でいいか、違う状態に切り替えるかを判断する

状態を返す

この状態の戦略からアクションを要求するためのメソッド

アクションを返す

これから、各状態のロジックを実装してAIを作ってみましょう。

AIの土台を作る

AIの実装を始めるために今のゲームの中に土台を用意しておきましょう。AIPlayerというクラスを作ります。初めの段階では、まだ何もする必要はありません。

```
var AIPlayer = Class.create(GamePlayer, {
     initialize: function(id, data) {
          GamePlayer.call(this, id, data);
     },

     simulatePlay: function() {
          this.controller.endTurn();
     },
});
```

シングルプレイのときはプレイヤー 2をAIPlayerにします。

```
beginCampaignGame: function(stageId) {
     中略
     if (this.getPlayer(2) == undefined) {   ← プレイヤー2が未定義なら
          var player2 = new AIPlayer(2, {name:"敵"});
          this.addPlayer(player2);
     }
     中略
},
```

さらに、ターン制御の中でAIPlayerのターンのときはsimulatePlayを呼びます。

```
startTurn: function() {
     中略
```

```
        this.updateTurn();
        if (player instanceof AIPlayer) {
            utils.beginUIShield();
            player.simulatePlay();    AIのロジックを実行
        }
    },
```

AIのアクションをゲームに反映する

ステートマシンからもらった判定をゲームに反映するのは`AIPlayer`のロジックの役割りです。

```
var AIPlayer = Class.create(GamePlayer, {
    initialize: function(id, data) {
        GamePlayer.call(this, id, data);
        this.state = new BaseState(this);
    },

    simulatePlay: function() {
        this.state = this.state.updateState();    状態の更新
        var action = this.state.chooseAction();    戦略の選択

        this.setActiveFune(action.fune);

        var self = this;
        setTimeout(function() {
            switch(action.type) {
                case "attack":    攻撃が選択された場合
                    action.fune.attackFune(action.target);
                    this.controller.endTurn();
                    break;
                case "skill":    スキル発動
                    var self = this
                    action.fune.activateSkill(function() {
                        self.controller.endTurn();
                    })
                    break;
                case "move":    移動が選択された場合
                    var self = this;
```

```
                        self.controller.map.moveFune(action.fune,
                        action.path, function() {
                            self.controller.endTurn();
                        })
                        break;
                }
            }, 1000);
        },
});
```

現代のCPUだとAIの処理は速すぎるので、自然に見えるため、わざと1000ミリ秒のあいだ停止させます

じつは、すでにこのロジックでAIと対戦することができます。もちろんほとんど戦略はありませんので、かなり適当な動きになりますが、これだけでも何となくAI対戦ができるようになるのです。

ステートマシンの実装 (Opening)

ステート変更の仕様

序盤では、とにかくマップをコントロールするため移動を繰り返します。相手が攻撃範囲に入ったら状態を中盤である`MidGame`（攻撃モード）に切り替えます（サンプルはchapter_19 → 37にあります）。

```
var OpeningState = Class.create(BaseState, {
    updateState: function() {
        var count = this.player.getFuneCount();
        for (var i=0; i < count; i++) {
            var fune = this.player.getFune(i);
            var targets = this.getTargetsWithinRange(fune);
            if (targets.length > 0) {
                console.log("AI switch from OpeningState to
                MidGameState");
                return new MidGameState(this.player);
            }
        }
        console.log("AI in OpeningState");
        return this;
    },
```

敵が攻撃範囲にいれば

状態を変更

基本戦略の流れ

防御系の船のスキルは効果が継続するので事前準備をかねてある程度の確率で発動するようにします。それ以外のスキルはあまり意味がないので使いません（低確率で発動するようにしてもよいのですが、ランダムに発動すると馬鹿なAIに見える恐れがあります）。

```
chooseAction: function() {
    var fune = this.getRandomFune();

    // Skill                防御系の船が選択された場合
    if (fune instanceof KataiFune) {          30%の確率でスキル発動
        if (fune.canUseSkill() && Math.random() < 0.3) {
            console.log("AI use skill", fune.getCaptainName(),
            fune.getSkillName());
            return {
                type: "skill",
                fune: fune,
            }
        }
    } else {
        序盤ではその他のスキルは発動させません
    }
```

基本的に大きく動くことを優先にします。

```
chooseAction: function() {

    行動が選択されなかったときは移動

    var moveList = this.getMovePosition(fune, 0.5);
    var pathList = [];
    for (var i=0; i < moveList.length; i++) {
        var targetPosition = moveList[i];
        if (targetPosition.i <= (fune.i +2) && targetPosition.j
        >= (fune.j -2)) {
            pathList.push(this.player.controller.map.getPath(fune,
            fune.i, fune.j, targetPosition.i, targetPosition.j));
```

```
            }
        }
        大きな移動を優先
        pathList = this.sortPathByLength(pathList);
        移動距離に30%のばらつきをもたせる
        var randomLongPathIndex =
        Math.floor(Math.random() *(pathList.length *0.3));
        var path = pathList[randomLongPathIndex];
        if (path == null) {
            console.log("AI no safe path");
            path = this.getRandomPath(fune, 0.0);
        }
        console.log("AI random long Move", fune.getCaptainName());
        return {
            type: "move",
            fune: fune,
            path: path,
        };
    },
```

ステートマシンの実装 (Mid-Game)

ステート変更の仕様

　中盤は戦いが始まる状況からスタートします。どちらかのプレイヤーの船が半分となったら終盤である**EndGame**に状態を切り替えます。

```
var MidGameState = Class.create(BaseState, {
        updateState: function() {
          勝っている場合
        if (this.player.getFuneCount() <=
        Math.ceil(this.player.getFuneCountInitial() /2)) {
            console.log("AI switch from MidGameState to EndGameBadState");
            return new EndGameBadState(this.player);
        }
```

負けている場合

```
    if (this.player.controller.getNonActivePlayer().getFuneCount() <=
    Math.ceil(this.player.controller.getNonActivePlayer().
    getFuneCountInitial() /2)) {
        console.log("AI switch from MidGameState to EndGameGoodState");
        return new EndGameGoodState(this.player);
    }
    console.log("AI in MidGameState");
    return this;
},
```

基本戦略の流れ

攻撃効果の高そうな船を優先的に動かすロジックを実装してみてもよいのですが、かなり複雑になる可能性があります。ここでは、船を順番に選択するロジックにしました。一見、いいかげんそうに見えますが、ランダムに選択されることで十分強い人工知能になっています。逆にこれ以上強くすると、人間が簡単に勝てなくなる可能性もあります。

```
chooseAction: function() {
    var count = this.player.getFuneCount();
    for (var i=0; i < count; i++) {
        var fune = this.player.getFune(i);
```

最初はスキル発動からチェックをします。

スキルを使う

```
var skillUse = this.testSkillUseInCombat(fune);
if (skillUse) {
    console.log("AI use skill", fune.getCaptainName(),
    fune.getSkillName());
    return skillUse;
}
```

スキルが発動しなかったときは、攻撃範囲内の相手から攻撃のターゲットを選びます。

```
chooseAction: function() {
      中略    ターゲットを探す

      var targets = this.getTargetsWithinRange(fune);
      if (targets.length > 0) {
          if (Math.random() < 0.7) {   攻撃する70%、攻撃しない30%
              var target = targets[Math.floor(Math.random() *targets.
              length)];
              console.log("AI attack from", fune.getCaptainName(),
              "on", target.getCaptainName());
              return {
                  type:   "attack",
                  fune:   fune,
                  target: target,
              }
          }
      }
```

選択している船のHPが50%以下になっているなら逃げることにします。

```
chooseAction: function() {
      中略    HPが50%以下なら逃げる

      if (fune.getHP() < (fune.getHPMax() * 0.5) ) {
          if (Math.random() < 0.3) {
              if (this.isTargeted(fune)) {
                          90%の確率で危険なマスを避ける

                  var path = this.getRandomPath(fune, 0.9);
                  if (path) {
                      console.log("AI escaping", fune.getCaptainName());
                      return {
                          type:"move",
                          fune: fune,
```

```
                            path: path,
                        }
                    }
                }
            }
        }
```

何もしなかったら序盤（Opening）のようにランダムに動くことにします。

これで中盤（mid game）の行動は実装できました、次は終盤（end game）の2パターンの実装を見てみましょう。

ステートマシンの実装：End-Game（勝ち状況）

ステート変更の仕様

相手の船の数が50%以下のときにこの状態になります。相手の船が多くなると`EndGameBadState`に状態が変わります。

```
var EndGameGoodState = Class.create(BaseState, {
    updateState: function() {
        if (this.player.getFuneCount() < this.player.controller.
        getNonActivePlayer().getFuneCount()) {
            console.log("AI switch from EndGameGoodState to
            EndGameBadState");
            return new EndGameBadState(this.player);
        }
        console.log("AI in EndGameGoodState");
        return this;
    },
```

自分の船と相手の船を比較

状態遷移

19

基本戦略の流れ

こちらのステートは`Opening`と`MidGame`の部分を再利用していて、新しいコードはありません。戦略

は次のようになります。

- **技が使えるなら使う**
- **相手に攻撃をする**
- **ランダムに動く**

ステートマシンの実装：End-Game（負け状況）

ステート変更の仕様

自分の船の数が50%以下のときにこの状態になります。自分の船の数のほうが多くなるとEndGame GoodStateに状態が変わります。

```
var EndGameBadState = Class.create(BaseState, {
    updateState: function() {
        if (this.player.getFuneCount() > this.player.controller.
        getNonActivePlayer().getFuneCount()) {
            console.log("AI switch from EndGameBadState to
            EndGameGoodState");
            return new EndGameGoodState(this.player);
        }
        console.log("AI in EndGameBadState");
        return this;
    },
```

自分の船と相手の船を比較

状態遷移

基本戦略の流れ

こちらのステートも、新しいコードはありませんので全体的な戦略を紹介します。

- **弱った船は逃げる**
- **技が使えるなら使う**
- **相手に攻撃をする**
- **ランダムに動く、基本的に相手に近寄らない**

対CPUの対戦モードも作っておこう

せっかくAIを開発したのですから、ストーリーモード用に作ったAIを対戦モードでも使えるようにしておきましょう。はじめに「VS」を選択したあと、ひとつ画面を追加して人間同士の対戦とCPU対戦を選べるようにしましょう。

選択により`beginVersusGame`で`GamePlayer`か`AIPlayer`をプレイヤー２に設定します。これだけで実装可能です（サンプルはchapter_19 → 38にあります）。

```
beginVersusGame: function(opponent) {
      中略
      var player2;
      if (opponent == "human") {
          player2 = new GamePlayer(2, {name:"プレイヤー2"});
      } else if (opponent == "ai") {
          player2 = new AIPlayer(2, {name:"プレイヤー2"});
      }
```

対戦相手がAIなら`AIPlayer`を使用

19

今回作ってみたAIは一般的なゲームでも利用できるようなAIです。しかし、実際に深く考えているわけではなく、複雑なこともしていません。しかし、ある程度成立するゲームの戦略は組み込まれているので(スキルの使いどころなど)、プレイしているとまるで自律的に考えているような感じすらします。乱数で多少変わった挙動をするようになっているので、サプライズもあります。

もちろんこれは奥の深いAIの世界のほんの入り口を見ただけにすぎません。本格的なAIを作るためには勉強しなくてはいけないことがたくさんありますが、あまり難しいことをしなくてもそれなりのAIが作れるのは体験できたかと思います。

本章で勉強すること

- 敵ユニットを追加しよう
- JavaScriptの継承機能を使って効率よく実装しよう
- ステージごとに登場ユニットを設定
- 設定は別ファイルで管理しよう
- 設定ファイルを変更して、ちょうどよい強さに調整しよう

AIとストーリーモードのための仕組みができましたので、今度はストーリーモードのためのゲームデータ（コンテンツ）を用意しましょう。

敵ユニット

ストーリーモードのために敵ユニットを追加することにします。ボス以外は通常の船より弱い手下にすることにしましょう。

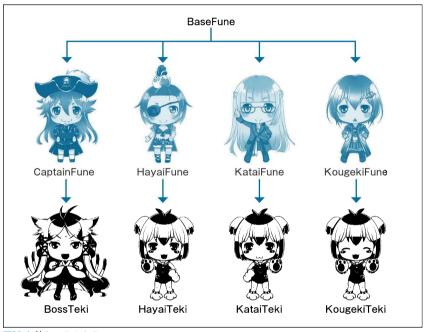



図20-1 敵キャラのクラスツリー

BaseFuneを継承して作った味方のユニットをもう一度継承し、見た目を変えます。継承することでAIの挙動を味方と同じにできます（サンプルはChapter_20 → 39にあります）。

```
var BossTeki = Class.create(CaptainFune, {     ← キャプテンの場合
    initialize: function(id) {
        CaptainFune.call(this, id);
                                                ← フレームを合わせる
        this.fune.frame     = [32, 32, ... 35];
        this.fune.sinkFrame = [35, 35, ... null];
    },
```

ほかの船についても同様の方法で作成しましょう。

```
var HayaiTeki = Class.create(HayaiFune, {    スピード系の場合
    initialize: function(id) {
        HayaiFune.call(this, id);

        this.stats.hpMax = Math.floor(0.5 * this.stats.hpMax);
        this.stats.hp = this.stats.hpMax;
                                                      フレームを合わせる
        this.fune.frame     = [32, 32, ... 35];
        this.fune.sinkFrame = [35, 35, ... null];
    },
```

これでユニットができましたので、**factory**クラスにも敵ユニットを追加をしましょう。

```
funeFactory: function(name) {    factoryクラスを改造
    switch(name) {
        case 1:
        case "captain":
        中略
        case 5:
        case "teki_boss":
            return new BossTeki (5);
        case 6:
        case "teki_hayai":
        中略
      }
    },
```

factoryができたので、ステージデータ上にこれらを簡単に配置できるようになります。

20

ステージデータ

ステージデータは手軽に編集ができるように別ファイルにしましょう。外部ライブラリなどのときと同じくHTML側に読み込みを追加します（サンプルはchapter_20 → 40にあります）。

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <meta http-equiv="x-ua-compatible" content="IE=Edge">
    <meta name="viewport" content="width=device-width, user-scalable=no">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <script type='text/javascript' src='../../../shared/javascript-astar/astar.
    js'></script>
    <script type='text/javascript' src='../../../shared/jStorage-master/
    jstorage.js'></script>
    <script type="text/javascript" src="../../../../build/enchant.js"></script>
    <script type="text/javascript" src="stageData.js"></script>   ここを追加
    <script type="text/javascript" src="main.js"></script>
    中略
</head>
```

ここでは今までのデータをそのまま外部のファイルにコピーしてしまいましょう。ファイルに出力された最終的なデータはかなり長くなりますので一部を紹介します。

```
(function() {
    window.StageData = [
        {
            startPositions: [
                {type: "teki _ hayai", i: 12, j: 0},   保存するデータ
            ]
        },
        {
            startPositions: [
```

20

```
            中略
            ]
        },
        中略
    ];
})();
```

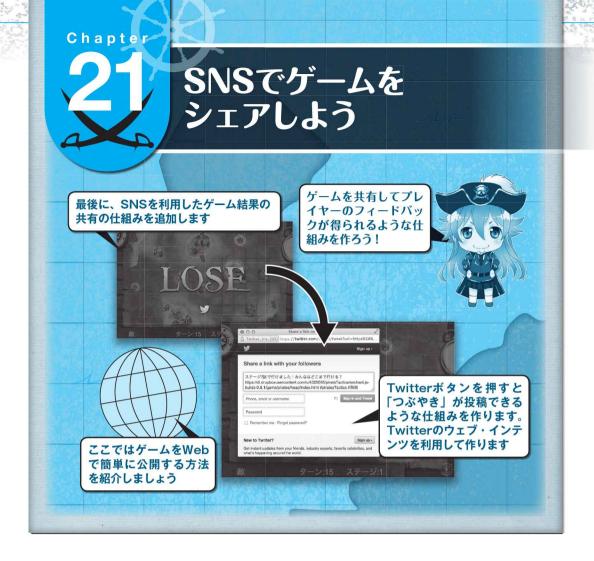
ステージのデザイン

実際のステージデータは長いので、ここではステージデータの概要を示すことにします（ステージ7以降はステージを繰り返すようになっています）。

▼ 表20-1 ステージごとの敵ユニット

ステージ	スピード系の敵	防御系の敵	攻撃系の敵	ボス
1	1			
2	1	1		
3		1	1	
4	2		1	
5	1	1	1	
6	2	1	1	
7	1	1	1	1

自分が失った船は失ったままになるので、けっこう難易度は高めですが、人工知能は完璧ではないので、頭と運を使えば勝つことは十分に可能です。自分でプレイしながら、最終的なバランスやステージデータを決めてみてください。

ラスボス戦！
bert/Desktop/pirateTactics/enchant.js-builds-0.8.1/game/pirates/39/index.html
プレイヤー1
ターン:0

本章で勉強すること

- ゲームはコミュニケーションツール。遊んでもらって真のゲームになる
- 誰でも気軽にゲームで遊べるようにゲームをネットに公開しよう
- ここではDropboxを使ったゲームの公開方法を紹介します
- ゲームの進捗はTwitterに投稿できるようにしよう
- 投稿はTwitterのウェブ・インテンツで実現
- ブラウザのウィンドウ操作にはJavaScript関数を使おう

お疲れ様です！ここまででゲームは完成しました！

でも、ゲームの本質はコミュニケーションであって、人に遊ばれることによって本当の意味でのゲーム

になります。

最後の章では、ここまで作ったゲームを誰でも遊べるようにするため、ゲームをシェア（配信/提供）する方法を紹介しましょう。

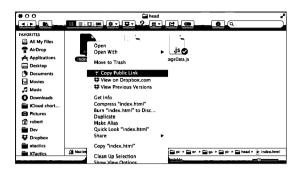
もちろん、ゲームにはいろいろな形があり、提供のしかたはそれぞれですが、今回はブラウザゲームをシンプルに共有できる手法を教えましょう。

ゲームをネットにアップロード

ネットでゲームを提供するためには、どこかのサーバーにアップロードする必要があります。自分でWebサーバーを立てるのはそれなりに手間がかかります。また、場合によってはお金もかかるでしょう。

しかし、最近はいろいろな無料サービスがあります。そのひとつとしてDropboxを使う方法を紹介しましょう。MicrosoftやAppleによる同様のサービスもありますが、Dropboxはどこでも使え、サービスも安定していますので今回はこちらを紹介します（もちろん、ほかのサービスで同じようにゲームをシェアしても大丈夫です）。Dropboxのアカウントを持っていない人は、最初にDropboxサイトでアカウントを作成してください。

DropboxのWebサイト
https://www.dropbox.com/

アカウントを作ったら、PC用のクライアントアプリケーションをダウンロードしてインストールしてください。そして、パブリックフォルダを有効にしてPirateTacticsのフォルダをまるごとDropbox/Public/の下に置いてしまいましょう。次に、Dropbox/Public/pirateTactics/enchant.js-builds-0.8.2-b/game/pirates/head/フォルダに移動し、メニューからIndex.htmlのURLをコピーします。

コピーしたら、あとはブラウザにURLをペーストして、Dropboxにゲームがアップロードされたかを確認します（ゲームがすべてアップロードされるのに数分かかる場合もあります）。

おそらく、ブラウザのURLは次のようになっているはずです。

https://dl.dropboxusercontent.com/u/*XXXYYYZ*/pirateTactics/enchant.js-builds-0.8.1/game/pirates/head/index.html

この「*XXXYYYZ*」の部分はDropboxのアカウント番号になっています。このURLをほかの人と共有すれば、基本的に世界のどこからでもゲームが遊べるはずです！

Twitterでシェア機能を作る

さきほどのURLをほかの人にも知らせたいので、Twitterでシェアしてみましょう。

今回、Twitterを使うには2つの理由があります。ひとつ目は、TwitterにはほかのSNSと比べて制約が少なく、ゲーム開発者がほかの開発者やプレイヤーと交流する場になっていることです。ふたつ目は、Twitterはプログラミングインターフェイスが公開されているため、ゲームと連動するための仕組みを組み込みやすいことです。

この本の読者であれば、Twitterについて知っている人も多いと思いますが、TwitterはSNS（Social Networking Service）のひとつです。140文字の短いつぶやき（Tweet）を気軽に投稿したり閲覧したりできます。SNSはおもに個人がネット上で交流するためのさまざまな仕組みを提供しています。人気のSNSとして、ほかにもLINEやFacebookの利用を考えてみてもよいでしょう。

Twitterでシェア機能を作る

ウェブ・インテンツの利用

今回は、Twitterのウェブ・インテンツ（Web Intents ）機能を使うことにしましょう。ウェブ・インテンツを使うとURLを利用して簡単にTwitterへの投稿が可能です。ウェブ・インテンツの使い方の詳細はTwitterの開発者向けサイトで解説されています。

https://dev.twitter.com/web/intents

英語の情報しかありませんが、今回はTweetを発信するだけですので、その部分の方法を簡単に説明します。

Tweetを発信するためのURLは次のようになっています。

https://twitter.com/intent/tweet

このURLに次のパラメータをくっつけてアクセスします。

url：シェアしたいURL
via：連携したいTwitterのアカウント（おもに自分のアカウント）
text：シェアしたい文章（最大140文字ですが、urlやviaなどもこの140に数えます）
hashtags：Twitterの#タグでシェアしたいキーワード

上のインテントを利用してTwitter上にゲームをシェアするためのボタンを作りましょう（サンプルはChapter_21 → 41にあります）。

```
var TwitterButton = Class.create(Sprite, {
    initialize: function(options) {
        Sprite.call(this, 64, 64);
        this.image = game.assets[uiTwitterBtnSprite];
        this.stageId = options.stageId;
        this.url     = options.url;
    },
    ontouchend: function(params) {
        window.open("https://twitter.com/intent/tweet?url="
            +encodeURIComponent(this.url)
            +"&text="+encodeURIComponent("ステージ"+this.stageId+"まで
            行けました！みんなはどこまで行ける？")
            +"&hashtags=piratesTactics,
            海賊", "twitter", "top=50, left=50, width=500, height=400");
    }
})
```


ここでは新たにブラウザのJavaScript関数を2つ使いました。それぞれ紹介します。

window.open(URL, ウィンドウ名 [,オプション])

実行すると新たにウィンドウが作られます。「オプション」の部分にウィンドウのいろいろな属性を指定します。複数の属性はカンマで区切って指定します。

▼ 表21-1 window.open関数に指定する属性

属性	説明	値
width	ウィンドウ幅	数値
height	ウィンドウ高さ	数値
left	ウィンドウ位置左	数値
top	ウィンドウ位置上	数値
menubar	メニューバー有無	yes/no
toolbar	ツールバー有無	yes/no
location	アドレスバー有無	yes/no
status	ステータスバー有無	yes/no
resizable	リサイズ可否	yes/no
scrollbars	スクロールバー有無	yes/no

encodeURIComponent

文字列変換の関数です。ブラウザのURLにはアルファベット以外の記号や日本語の文字を直接使うことができません。そこで、このメソッドでURLで使える文字に変換します。

●記号の変換

http://www.google.co.jp/search?hl=ja&q=encodeURIComponent

↓

http%3A%2F%2Fwww.google.co.jp%2Fsearch%3Fhl%3Dja%26q%3DencodeURIComponent

●日本語の変換（「日本語」という文字列を変換しています）

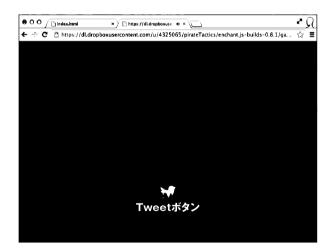
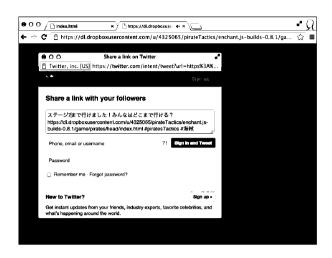
日本語

↓

%E6%97%A5%E6%9C%AC%E8%AA%9E

UIの作成

あとはストーリーモードで負けたとき、`GameManager`で`TwitterButton`を生成し、保存されているステージIDと自分のDropboxのURLを渡します。

```
campaignOver: function(winner) {
    中略
    if (winner.id == 1) {
        勝ったときの処理
    } else if (winner.id == 2) {
        resultBanner.image = game.assets[uiLose];
        var tweet = new TwitterButton({
            stageId: self.stageId,              これは自分のDropboxのURL
                url: "https://dl.dropboxusercontent.com/u/XXXYYYZ/
                     pirateTactics/enchant.js-builds-0.8.1/game/pirates/
                     head/index.html"
        });
        touchable.addChild(tweet);
        tweet.x = 480 -32;
        tweet.y = 450;
```

画面の中のTwitterの鳥のボタンを押したら、新しいウィンドウが開いて、Twitterのウェブ・インテンツが現れます。

これでプレイヤーが自分の進捗をTweetでき、周りの人にゲームのURLを知らせることで、遊んでくれる人を増やしていけます。

あとはどんどん自分のアイデアや新しいルールなどをゲームに取り込み、自分なりのステージやキャラをアレンジしてみてください。本書のソースコードも自由に変えてみて、学びながら自分のオリジナルなゲーム作りにチャレンジしてみましょう。

自分がゲームを作るのを楽しめば、ほかの人もゲームを遊んで楽しむようになっていくにちがいありません。

索引 —INDEX—

<は>

<ま>

<や>

<ら>

<わ>

著者プロフィール

思い出に残ったゲーム
『Crossroads(by Steve Harter)』
ほとんどの人間は聞いたことがないと思うけど、自分はこのゲームを雑誌のコードから入力して、ゲーム作りの勉強を始めました。

ロバート・ジェイ・ゴールド
(Robert Jay Gould)

アメリカ生まれ。5歳のクリスマスにComodore64というコンピューターを買って貰い、8歳から趣味でゲーム開発を始めたギーク。その後中学時代はアナログやテーブルトップゲームにも夢中になり、「人間がゲームを楽しむ」観点でのゲーム作り全体に興味が湧いた結果、大学では動物行動学と人工知能を学ぶ。その後は院生として日本に留学して、卒業後はセガ、スクウェア・エニックス、グリーでゲーム作りを経験。それでも物足りなさを感じ、このたび沖永とGAMKINを立ち上げた。

『EVE burst error』
やっぱり自分が業界を意識し始めた作品というと、これになりますかね。今でも一番心に残ってますね。

沖永賢吾
(Kengo Okinaga)

大学在籍中、ゲームデザイナーを目指し、就職活動するも遭えなく断念。自身の特性を考え、業界入りの可能性を模索した結果、株式会社カプコンの営業管理部門での入社を果たす。新卒一年目、アミューズメント施設運営業務に従事し、業務の傍ら上司にゲームデザイナーになりたいことを相談するが、入社職種が違うためになれないという現実を突きつけられる。その後、アニメ、PCオンラインゲーム、ソーシャルゲーム業界を経て、今まさに自身の人生すべて受け止め、2013年9月GAMKIN株式会社を設立し、ゲームデザイナーとして再スタートを切っている。

GAMKIN 株式会社

10年来の友人であった沖永氏とロバート氏が、2013年9月に設立したゲーム開発会社。両氏の「新しい体験ができるゲームを作りたい」という想いに賛同し、ゲーム業界の変遷と共に第一線で活躍してきた熟練のゲーム開発者達が集う。現在は、自社企画『X-Tactics』(クロスタクティクス)を全世界展開にむけて鋭意開発中。

▶http://www.gamkin.com/